Design Automation and QoS Requirements Preservation for Multiprocessor Embedded Systems

by
Md. Al Maruf

A thesis submitted to the
School of Graduate and Postdoctoral Studies in partial
fulfillment of the requirements for the degree of

Master of Applied Science in Electrical and Computer Engineering

Faculty of Engineering and Applied Science
Ontario Tech University, Canada

July 2019

THESIS EXAMINATION INFORMATION

Submitted by: **Md Al Maruf**

Master of Applied Science in Electrical and Computer Engineering

Thesis title: Design Automation and QoS Requirements Preservation for Multiprocessor Embedded Systems

An oral defense of this thesis took place on June 12, 2019 in front of the following examining committee:

Examining Committee:

Chair of Examining Committee	Dr. Walid Morsi Ibrahim
Research Supervisor	Dr. Akramul Azim
Examining Committee Member	Dr. Qusay Mahmoud
Thesis Examiner	Dr. Patrick Hung

The above committee determined that the thesis is acceptable in form and content and that a satisfactory knowledge of the field covered by the thesis was demonstrated by the candidate during an oral examination. A signed copy of the Certificate of Approval is available from the School of Graduate and Postdoctoral Studies.

ABSTRACT

The number of processors is increasing in embedded systems but the usefulness of parallel computation is not better leveraged due to the inflexibility of design and implementation for multiprocessor embedded system applications. A higher level abstraction (i.e., a parallel programming framework) can ease the programmers to define parallelism for tasks in an application, but designers still face the complexity of mapping high-level requirements to the syntax and semantics of a parallel programming interface. Nevertheless, embedded system applications execute various periodic tasks that are carried out repeatedly within a certain time interval. Each task is characterized by its deadline where it is expected to perform a function producing a correct result within a specified amount of time. These tasks may be able to run in parallel to utilize the system efficiently. Moreover, embedded systems often interact with dynamic environments requiring not only to meet deadlines of tasks but also to achieve a certain level of accuracy as the inaccuracy of a task output produces a similar adverse effect like timing violation. Therefore, it requires an automated design process to map the tasks to lower level and a monitoring framework to meet the high-level requirements of embedded system applications.

This thesis presents a parallel loop-based task construct to automate the design process of embedded applications from Architecture Analysis and Design Language (AADL) models and demonstrates how to preserve the requirements at lower-levels. AADL is practiced to model the software and hardware architecture of embedded systems. Since most of today's embedded systems either belong to soft real-time systems (i.e., stream processing systems) or weakly-hard real-time systems (e.g., control systems), we adopt a new task scheduling approach in a well-known parallel programming interface called OpenMP for increasing determinism in soft real-time system (RTS) applications. Moreover, we propose a calibration framework to increase the robustness of weakly-hard real-time system applications that rely on time-driven scheduling approaches such as rate monotonic (RM) scheduling. In this thesis, determinism and robustness are the way of measuring the quality of service (QoS) requirements of tasks. For increasing the robustness of weakly-hard real-time systems, the calibration framework is used by which the system component's output accuracy can be monitored and compared with a calibration standard. The calibration standard is derived from a representative component's output with known high accuracy. As an example, we analyze the accuracy of a component that performs dynamic voltage and frequency scaling (DVFS) and explains the associated timing effects in terms of task schedulability.

To illustrate the applicability of our mechanism, the experimental analysis incorporates a design automation process for mapping tasks to parallel programming framework in soft RTSs, a calibration framework for monitoring task output in weakly-hard RTSs. To understand the calibration framework, a software-based monitoring approach is shown for a resistive voltage divider as a case study. Therefore, we use a cost estimation model to demonstrate the efficiency of the automation process and map tasks over multiple processor cores using OpenMP. To ensure meeting high-level requirements of embedded system applications, we analyze the existing OpenMP scheduling mechanisms and propose a layer of adaptation. We show that our proposed adaptation layer facilitates a tighter execution time bound for time-sensitive tasks or a better throughput for tasks that require higher QoS. Thus, the proposed design automation framework is applicable for a variety of applications with different QoS requirements preserved at the lower level. To monitor the QoS of task output, we perform experiments on LITMUS$^{\mathrm{RT}}$ kernel to demonstrate the need and applicability of our calibration framework in the domain of embedded systems. The experimental results illustrate that the proposed approach yields more predictability and show better performance for preserving QoS requirements of tasks.

AUTHORS DECLARATION

I hereby declare that this thesis consists of original work of which I have authored. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize the Ontario Tech University to lend this thesis to other institutions or individuals for the purpose of scholarly research. I further authorize Ontario Tech University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research. I understand that my thesis will be made electronically available to the public.

Md. Al Maruf

STATEMENT OF CONTRIBUTIONS

The contributions of my work are listed as follows:
a) We present a design automation process for mapping and preserving high-level requirements in soft and weakly-hard RTSs. We propose a loop-based task construct to map periodic tasks into a parallel programming framework (i.e., OpenMP) in soft RTSs where the requirements are extracted from AADL model specifications. These requirements are considered as high QoS requirements or low QoS requirements.
b) In the case of weakly-hard RTSs, we present a calibration framework using LITMUS^{RT} which monitors the task requirements that are reliant on the accuracy of the system components. Moreover, it will guide to take action after comparing the output with standards to ensure the expected component's output accuracy.
c) In another work, we present a case study of a resistive voltage divider for a software-based auto calibration framework in the absence of a proper calibration standard. It examines the outputs of the voltage divider to find anomalies and notifies the system when the output goes out from the expected result.
A Part of the above the contributions has already been published as:

- Md. Al Maruf, and Akramul Azim. "Software-based Monitoring for Calibration of Measurement Units in Real-time Systems." IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society. IEEE, 2018, USA.

ACKNOWLEDGEMENTS

I would first like to express my heartiest gratitude to the Almighty Allah for giving me the strength to finalize this thesis.

I want to express the most profound appreciation to my supervisor Dr. Akramul Azim, Assistant professor, UOIT for his useful comments, remarks and engagement through the learning process of this master thesis. He provided me invaluable guidance, support and immense inspiration towards the completion of this thesis. The door to Dr. Akramul Azim office was always open for me, and I always ran toward his office whenever I faced any problem in my research. He consistently allowed me to work in my own interest and provided me the right direction when it was required. Thanks for believing in me and helping me in all aspects of my thesis work.
Furthermore, I would like to thank all of my RTEMSOFT research group members who have supported me throughout the entire process, both by helpful criticism sharing thoughts to improve my work. Many thanks to the members of the Software System Research Lab who contributed to the friendly atmosphere in my workplace. I also appreciate the support of all my friends, colleagues, and relatives who always motivated and encouraged me through this journey. I want to acknowledge the helpful assistance and instructions from the respected faculty members and staffs of Ontario Tech University.

Finally, I must express my very profound gratitude to my family, especially my beloved mother Mahfuza Begum, my father Motalleb Hossain, my sister Khairun Nesa, and my brother Naim Ali for their unconditional support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them. Thank you.

Contents

List of Figures

List of Tables

Abbreviations

RTS	Real-Time System
RTSs	Real-Time Systems
AADL	Architecture Analysis & Design Language
UML	Unified Modeling Language
OpenMP	Open Multi-Processing
QoS	Quality of Service
MDA	Model Driven Architecture
NIST	National Institute of Standards and Technology
RM	Rate Monotonic
EDF	Earliest Deadline First
MC	Mixed Criticality
DVFS	Dynamic Voltage and Frequency Scaling
COCOMO	Constructive Cost Model
LOC	Line of Code
TAR	Test Accuracy Ratio
DUT	Device Under Test
MILNP	Mixed Integer Nonlinear Programming
API	Application Programming Interface
CPS	Cyber-Physical Systems

Chapter 1

Introduction

Real-time embedded systems are those systems that require to respond within strict time constraints and provide a worst-case time estimate for critical situations. Typically, real-time embedded systems are classified into two types, hard and soft RTSs. The hard RTSs consider any missed deadline to be a system failure. A deadline of a task is a specified time indicating when a task to be completed. On the other hand, a soft RTS allows a task to miss the deadline, but the system service is degraded. However, we usually see most of the hard RTSs can tolerate a certain delay that is defined by the system user, not allowing to cross the delay limit. Therefore, these systems are called weakly-hard RTSs [3][4]. For example, a control system application can tolerate some delay during its execution. In this system, the distribution of its task requirements and missed deadlines during a window of time is precisely bounded. The embedded system applications differ from each other in terms of their design and requirements. The requirements are defined as timing requirements and QoS requirements. The QoS requirements are translated as timing requirements. The high QoS requiring tasks need deterministic execution where the other tasks may not always require deterministic execution. A system needs to satisfy those requirements by computing application tasks into different platforms.

To meet the increasing demand for high-performance computing, the use of multiprocessor systems is growing rapidly. The design of embedded system applications in these multiprocessors is becoming very complex due to the integration of various applications into a single platform. However, the programmers experience many implementation challenges (e.g., mapping application tasks, changing performance requirements) while writing parallel programs for multiprocessors. In order to overcome the implementation challenges [5], many programming frameworks enable execution of tasks in parallel

by substantially raising the level of abstraction in implementation. Although many programming frameworks exist, the application designers face difficulties in mapping specifications to tasks with varying requirements [6].

Moreover, an embedded system executes a number of tasks on different components (e.g., hardware and software) where each task is required to complete its operation within a stringent timing deadline utilizing limited computing resources. In any dynamic environment that changes frequently, a system including integrated components requires robustness in generating the correct output. The robustness requires a system or component to operate correctly in the presence of invalid input or stressful situation [7]. The output of a task requires to represent a good QoS of the system or components. Therefore, embedded systems need to be implemented carefully to automate the design process that maps high-level requirements and monitors the task's output for meeting the requirements of the parallel tasks.

1.1 Design automation of parallel periodic tasks

Embedded system applications execute different types of tasks [8] that are hard real-time, soft real-time, and non real-time. In this thesis, we assume tasks that are not hard real-time but may enforce timing requirements weakly to achieve a high QoS. Moreover, embedded system applications execute tasks periodically for a long time until the system is out of service. This requires application tasks to execute repeatedly with predictable execution times. Therefore, embedded systems with soft real-time tasks require a parallel programming environment to run independent tasks repeatedly in predictable times.

As an example, a home automation system may have multiple applications that periodically monitor a predetermined area for an emergency state comprising various tasks like smoke detection, carbon monoxide (CO) detection, capturing images, and measuring temperature. Each of the tasks is scheduled to run repeatedly after every fixed time unit. This repeated task execution requirement can be controlled using a loop.

1.1.1 Mapping of AADL specifications to a loop-based task construct

In any embedded system application comprising of periodic tasks, each task requires to run concurrently to assure the better safety of the monitoring area. To achieve task-level parallelism upon a multiprocessor platform, we can execute the multiple independent tasks at the same time and accelerate the performance of the system [5]. However,

such requirements and the parallel programming interface syntax often impose various design constraints to automate the high-level application requirements as defined in specification languages such as Architecture Analysis and Design Language (AADL) [9]. An AADL specification defines various kinds of software and hardware component types such as systems, processes, threads, processors, and buses.

In this thesis, we propose a loop-based task construct that helps us to map the high-level requirements of periodic tasks into task constructs of a parallel programming interface. We see the increasing use of multiprocessing API called OpenMP to overcome the design and implementation challenges [10] in soft RTSs. OpenMP is a parallel programming interface that offers loop-based language constructs in addition to the thread-based execution abilities by `for loop` iterations. This seems a perfect fit for running soft RTS applications because of the repeated behavior of tasks. Thus, in this work we present an automatic conversion of our AADL-based task constructs into the loop-based task constructs for OpenMP in soft RTSs. Besides, we use the loop based task construct to run in weakly-hard RTSs and monitor the task output for calibration in LITMUS^{RT}.

1.1.2 Task scheduling

We analyze different scheduling approaches for parallel periodic tasks which are available in OpenMP and LITMUS^{RT}. OpenMP offers different scheduling algorithms to run the tasks in the system. It is essential to analyze the performance of execution times for different scheduling algorithms because of the varying requirements of embedded system applications. Therefore, in this design approach, we focus on soft real-time embedded applications with requirements of the high and low QoS, leaving the extension to other types of systems in the future. We perform a detailed analysis of static and dynamic [10] approaches offered by OpenMP to analyze the differences in execution timing behavior. Moreover, we examine the static, dynamic and rate monotonic scheduling approach to provide better throughput and response time for tasks focusing on QoS [11, 12].

1.2 QoS Requirements preservation

To preserve the application requirements, our designed framework includes an adaptation layer that guides to select the right scheduling policy based on the requirements of tasks in soft RTSs and a monitoring framework to ensure the expected output in weakly-hard

RTSs. Therefore, we explain how the QoS requirements are adapted and what challenges exist toward the implementation of the adaptation layer.

1.2.1 Requirements adaptation layer

In our proposed design approach, we achieve a tighter bound on execution time on multiple runs of the same task by binding program execution threads to processors. We also provide flexibility in choosing the scheduling approaches for different requirements. The designed framework includes an adaptation layer where we can define different configurations to meet the high-level requirements of applications. In this work, we assume two types of requirements for tasks which are of high QoS and low QoS. As an example, in an obstacle detection avoidance scenario, a robot requires to get all the updated data (e.g., image) from all the sensors. Therefore, the communication for transmitting data for detecting and avoiding obstacles define the QoS of the task. For instance, whenever a robot follows a straight line, the obstacle detection sensors should transmit the data at the highest rate for detecting and avoiding obstacle where the line sensors can be scanned at a lower data rate. As a result, the obstacle detection task demands a high QoS requirement by maintaining high transmission of data within a certain time. On the contrary, the line follower data transmission at a lower rate can be leveled as a low QoS requirement as it is following a straight line. These QoS requirements are interpreted as timing or output accuracy requirements. Therefore, we consider QoS requirements for soft and weakly-hard RTSs in our thesis work. Figure 1.1 shows the abstract view of our thesis workflow for different RTSs.

1.2.1.1 Satisfying QoS requirements for soft RTSs

In the case of soft RTSs, a task is expected to complete within a certain time but failing to meet the requirements does not have any catastrophic failure. Therefore, the use of the OpenMP parallel programming framework to implement such soft RTS applications can help to automate the design process and meet the QoS requirements more adequately. The high QoS requirement refers to meeting timing requirements predictably where the low QoS requirement does not require to run deterministically due to the best effort execution behavior. To meet the high QoS requirement of tasks, we propose the OpenMP static scheduling approach with the addition of using processor affinities. The thread to processor binding design makes it possible to assign tasks into processors

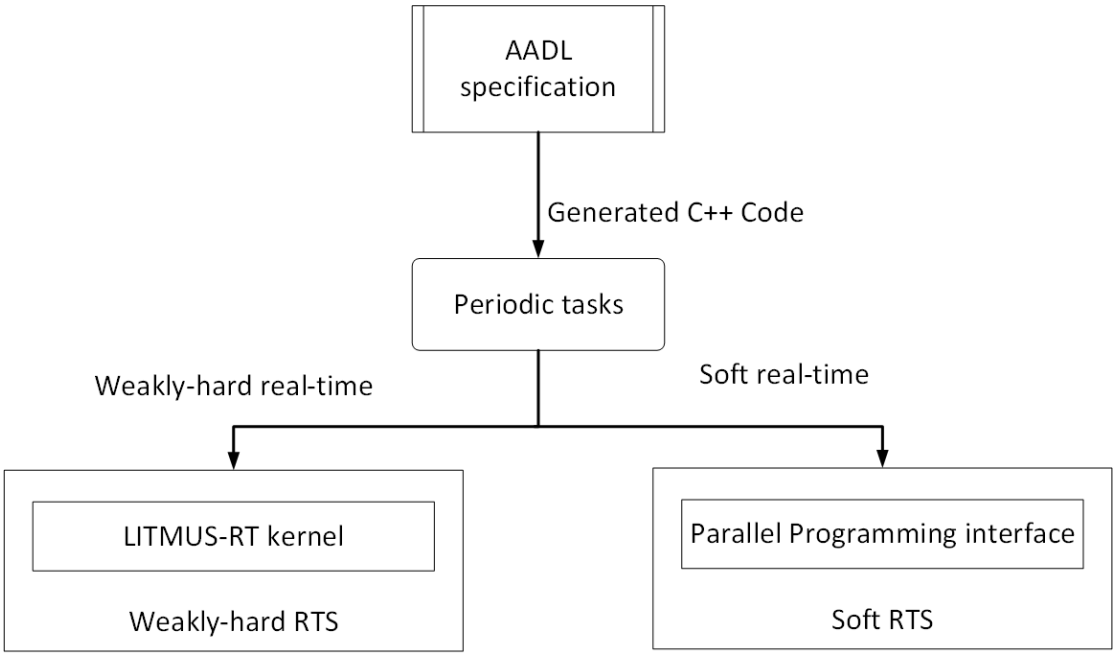


FIGURE 1.1: An abstract view of our design automation and requirements preservation workflow

deterministically because of controlling a task where it will execute. Such a fixed assignment of tasks to processors reduces the number of migrations among processors. Binding tasks to threads for mapping to processors during scheduling takes the advantages of data locality and therefore minimizes the synchronization overhead significantly. This results a tighter bound on execution time for tasks executed repeatedly. On the other hand, we use the dynamic scheduling approach for running low QoS requirement of tasks that require faster execution.

1.2.1.2 Satisfying QoS requirements for weakly-hard RTSs

To preserve the QoS requirements in weakly-hard RTSs, we present a monitoring approach using a real-time kernel (LITMUS^{RT} [13]) and partitioned multiprocessor Rate Monotonic (RM) scheduling algorithm. This monitoring approach monitors the accuracy of system components to ensure the expected output concerning the QoS requirement. Therefore, we propose a calibration framework that monitors the component's output accuracy and guides to take action if the output deviates from the expected result. To understand the accuracy-related QoS requirements, we discuss first the basic terminology regarding the component's accuracy in embedded systems.

A component of a system deteriorates over time, and therefore the performance degrades gradually. Different uncertainty factors such as aging of the software of a component, environmental changes, normal wear and tear of the hardware involving in a component, and process changes can cause deterioration on the accuracy. Therefore, a deteriorated system component produces a deviated output that includes various offsets, drifts, and noises. This results in the component's output accuracy that determines the correctness of output compared to a standard value. In embedded systems, the fulfillment of real-time requirements depends on a certain level of accuracy. For example, the electronic speed control in any aircraft needs to produce a correct speed before its landing, otherwise an incorrect output while measuring speed can lead to an undesirable consequence.

In order to ensure an expected accuracy, automatic periodic calibrations are performed to estimate error in output against the known calibration standards (e.g., NIST) [14] and corrections are made to minimize errors. In measurement science, calibration is a way of comparing the measurement values delivered by a component or device under test with a standard calibration device that has a known accuracy. In practice, calibration also includes output error adjustment of the component if it is out of the accepted range of accuracies. The standard calibration components are assumed to provide correct measurements. Most of the system components are calibrated against the highest level of standards such as international, national, primary, secondary and laboratory standards. However, in the case where such externally defined standards are absent, a user-defined standard can also be developed by different methods such as log correct output or trace data patterns taken from a high accuracy system component. To ensure a fair measurement of accuracy in the calibration process, different measurement communities recommend maintaining a test accuracy ratio (TAR) $4:1$ [15], which is the ratio of accuracies between a standard and a unit under test. The ratio of TAR means the standard components should be four times more accurate than the unit under test.

The QoS degradation of a component due to inaccuracies may cause a task to miss the requirements (e.g., deadlines). For example, energy-aware scheduling algorithms use dynamic voltage and frequency scaling techniques where the output voltage supply provided by a voltage divider is required to maintain a certain accuracy to guarantee a task deadline. However, an incorrect measurement in producing the voltage supply causes an undesired clock frequency. Figure 1.2 shows how a task execution time changes

while we modify the supply voltage for different CPU clock frequencies. This eventually provides an execution time delay for which a task may miss the deadline and a deadline miss may lead to a component or system failure eventually. To avoid such occurrence, it is required to perform calibration, but we need to analyze when a component needs it.

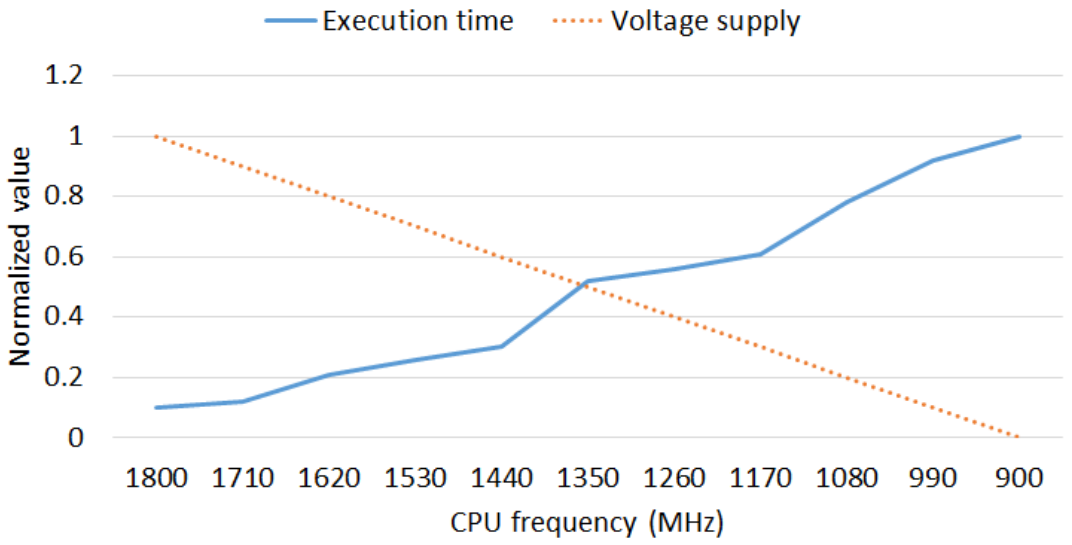


FIGURE 1.2: Execution time changes for different CPU frequencies [1]

1.2.2 Monitoring task output for calibration

In this thesis, we propose a calibration framework for weakly-hard real-time systems to determine whether we need to take any recovery actions of a system component immediately or we can wait by leveraging the low utilization of a system to delay the recovery process which is usually expensive. The laxity on the timing requirements of tasks is considered without compromising any safety requirement (i.e., deadline misses). Unlike the existing studies, we present a calibration framework that aims to increase the robustness of a system by monitoring the correctness of measured values and reviewing task schedulability. Moreover, we allow provisioning of the increase in the execution time and the deadline of a task based on the available CPU utilization and a user-defined tolerable delay. On the other hand, a schedulability test is proposed with the consideration of the accuracy of a system component using the rate monotonic task scheduling. Moreover, the component's output accuracy affects the QoS of a task. Thus, our proposed framework records all the related system input, output, and scheduling traces upon a task execution to monitor the component accuracy and measure the quality of a task output. To perform calibration, our experiment in LITMUS^{RT} defines the initial

trace data of a component as a calibration standard and shows the necessary steps that are required to correct the output.

1.3 Challenges in requirements preservation

The design and implementation of embedded system applications with varying requirements are challenging. The implementation requires the proper design in which area the requirements should be handled. On the other hand, the designers face challenges in mapping requirements to any implementation approach. Such kind of interactions often makes it more complex to automate the development process of embedded software.

1.3.1 System-level formal specification

The high-level specification of embedded system application is mainly based on informal requirements. Most of the cases, it does not fulfill the actual requirements. Moreover, it is challenging to design any application for multiprocessor systems. Assigning tasks at runtime and addressing requirements according to specification introduces difficulties for both programmers and designers. As a result, the designer and programmer have confusion regarding the mapping of requirements at design and implementation levels. However, it is also difficult to specify the whole specification at the very beginning due to the limitations of the programming language barrier, time, security, cost and performance issues [16]. Therefore, it requires design automation to ease the development process in embedded systems.

1.3.2 Embedded software complexity

Nowadays, the number of lines of source code for developing embedded system applications is growing with multiple requirements of customers. As a result, embedded software takes the maximum budgets of implementation. Moreover, the design complexity of the software component causes serious concerns for the final application [16]. In general, the design of an embedded system requires verification of the system and the long redesign cycles make it more complex in the development process.

1.3.3 Requirements integration

In embedded system design, we observe several domain-specific modeling languages (e.g., UML, AADL) that are practiced for different purposes considering performance, expertise, cost, safety, and security, etc. For example, UML (Unified Modeling Language) is known as a general modeling language and SysML (Systems Modeling Language) is used in systems planning related applications. However, AADL is used for many embedded software architecture designs to handle different high-level requirements. Moreover, it illustrates the system constructs such as threads, processes or processors to model real-time, safety-critical embedded systems. Moreover, it provides an advantage to model embedded system applications by analyzing task schedulability, and safety characteristics. The automatic code generation from AADL specifications drastically optimizes the executable code writing. System design is the process of deriving, from requirements, a model from which a system can be generated more or less automatically.

1.3.4 Selection of a task scheduling approach

We study different scheduling approaches that have their own benefits in different scenarios. The OpenMP parallel programming framework offers dynamic and static scheduling approaches. The current static approach in OpenMP runs reasonably in guaranteeing predictable execution times for deterministic applications but it shows non-repeatability behavior in producing a similar result on multiple runs and fails to meet a predictable execution [17]. Thus, finding a way to ensure the deterministic execution of parallel programs is also an important research question for us in this work. Similarly, the weakly-hard real-time systems require the real-time kernel to ensure the timing guarantee. As an example, the LITMUS^{RT} kernel provides different scheduling approaches like Earliest Deadline First (EDF) and Rate Monotonic (RM) where partitioned multiprocessor RM scheduling approach provides more reliable condition to meet the deadline of higher priority tasks over EDF approach.

1.4 Thesis objectives

To automate the design of embedded software, we find a variety of abstraction levels, execution platform, implementation approaches, and application requirements that are

not supported by only one development tool. As a consequence, neither a design approach nor a single parallel programming framework supports the diversity of embedded software implementation. The multiparadigm modeling approaches are performed to integrate the design model and automate the code generation for implementation. Therefore, the main objective of this thesis is to automate the design process for mapping high-level requirements to lower-level implementation. During the automation process, we meet the requirements of the application at different levels by increasing substantial abstraction in soft and weakly-hard RTSs.

1.5 Contributions

In summary, the main contributions of this thesis are two-folds:

1. **Parallel Loop-based Task Construct:** We automate the multiprocessor embedded system application design using a parallel loop-based task construct from AADL models that map the tasks high-level requirements and their properties to the semantics of a parallel programming interface.

2. **Requirements Preservation:** To satisfy various tasks requirements, we include an adaptation layer that helps to satisfy tasks requirements without making any changes at the operating system level. In this work,

    - we propose to bind the program execution threads to processors in OpenMP static scheduling to achieve a tighter bound on the deterministic execution time for high QoS requirement of tasks but use OpenMP dynamic scheduling for low QoS requirement of tasks. These requirements are adapted through the OpenMP parallel programming framework in soft RTSs.
    - we also present a calibration framework that monitors the output accuracy of components in weakly-hard RTSs. It includes an accuracy-based task schedulability test that guides to calibrate a system component when required.

1.6 Organization of the thesis

We organize the thesis in five chapters. In Chapter 2, we discuss the design automation and requirements preserving related works and basic terminologies that will help to

follow the remainder of our thesis work. This chapter contains the recent design automation techniques; task scheduling approaches in multiprocessor and different monitoring approaches including calibration for ensuring the correct output of embedded systems. In Chapter 3, we demonstrate the workflow of our proposed design automation framework and discuss the proposed approaches to preserve requirements. The proposed approach considers the QoS requirements preservation in soft and weakly-hard RTSs. The calibration framework monitors the system output to ensure the expected accuracy. Moreover, we show an illustrative example of how a software-based calibration can be performed when the calibration standards are not available. In Chapter 4, we present the experimental analysis of the thesis that has different sections for understanding the advantages of our proposed approaches. The experimental results include the analysis of requirement preservation approach using OpenMP in soft RTSs and output monitoring for calibration in weakly-hard RTSs. Finally, Chapter 5 concludes the thesis and points out possible future research directions in the context of requirements preserving design automation of embedded software.

Chapter 2

Literature review

2.1 Introduction

The architecture of multiprocessor is becoming more attractive for embedded systems. Embedded systems can be classified into three types which are hard, weakly-hard and soft. The hard real-RTSs are not allowed to miss the deadline of tasks. On the other hand, weakly-hard and soft RTSs have some flexibility in terms of deadline miss. However, the current hard RTSs are designed in such a way so that it can tolerate a certain level of permissible delay. These are often called weakly-hard RTSs.

Moreover, the current soft and weakly-hard RTS applications comprise of varying requirements that are increasing every day. To satisfy varying requirements, the design of embedded systems becomes more complicated as the number of components, implementation classes, modules, and methods are integrated into a particular system [18]. Although there exist a lot of works to design systems on a uniprocessor, the number of attempts for multiprocessor embedded systems are not much explored. We examine a few parallel programming frameworks for writing parallel programs but the requirements mapping in these frameworks requires a lot of efforts. Existing parallel programming interfaces are also not suitable for hard or weakly-hard RTSs. To address this issue, the structure of the abstraction level for mapping requirements of tasks is continuously advancing. A few works show the different techniques to automatic code generation but the requirements are not preserved properly. Therefore, the development process for embedded systems design from high-level requirements requires a framework to preserve requirements in design automation.

2.2 State of the art of design automation and requirements monitoring

An embedded system is a computing system that is designed to execute different functions. In recent time, the properties of hardware, software, and environments are changing rapidly. As a result, the embedded system design requires different techniques for predictable execution and adaptation of system requirements. The current practices for embedded system designs are:

2.2.1 Language-based design

In this approach, the embedded software design follows either traditional software implementation methods or synthesis-based methods that evolve from hardware design methodologies. A classic language-based design supports a specific programming language for a targeted system. As an example, Ada and RT-Java languages.

2.2.2 Model-based design

In recent trends, the hardware and software specifications are combined to design the embedded systems. This approach has substantial control over the other design approaches as it emphasizes the separation of two different levels (e.g., design level and implementation level). Due to the independence of two levels, the semantics of abstract system also increases. The model-based approaches mainly focus on producing an efficient code generation as it requires to implement sperate code for hardware and software components. A simplified model-based approach is the use of MATLAB Simulink which includes the simulation engine to define different operational properties [19][20].
However, the recent modeling languages are UML and AADL that emphasis on system architecture as a means to organize computation, communication, and constraints. UML is commonly used for the implementation of embedded system applications and provides an opportunity to generate source code for both hardware and software.
Similarly, the AADL specification assists in the design process by identifying design errors before any application implementation. Moreover, it helps to generate high-quality AADL code that can lower the development costs. The advantages of using AADL in automation are listed below.

- AADL provides a standard format with correct syntax and semantics.

- It presents the software architectures overview which can be modified to integrate different properties of applications. For example, task schedulability, safety, communication latency, hardware components, software components and so on.
- It enables advanced tracking of modeling and analysis. The specification includes the system structure and runtime behavior.
- It supports code reusability in implementation approach.

However, we find a few tools to convert the AADL specification to expected program code but Ocarina [21] is one of them. It analyzes the AADL specification and produces the code for application implementation. Since it has a modular architecture, it can support customized functionalities to use its existing codes. The main features of Ocarina are:

- Parser: Ocarina supports both AADL1.0 and AADLv2 and able to parse syntaxes to a particular language.
- Code generation: The conversion of code is mainly targeted for C real-time OS.
- Model checking: Ocarina can map AADL models onto Petri Nets.
- Schedulability analysis: It also provides a mapping of AADL models onto Cheddar that analyzes the real-time performances.

2.2.3 Implementation approaches in a multiprocessor platform

With increasing demands of high-performance computing and different requirements, the use of multiprocessors embedded systems is increasing. In recent works, we see the programmers use different parallel programming framework to run the applications in parallel. However, the designers face challenges in mapping high-level requirements to a parallel programming interface. Existing approaches like model-based automotive partitioning and mapping [22] use the AMALTHEA tool platform in developing automotive embedded multiprocessor systems. Through the interfaces of AMALTHEA, a user can develop different AUTOSAR (Automotive Open Systems Architecture) applications. Although the model-based design addresses various challenges considering the parallel exploitation, it can be further extended to automate the high-level requirements mapping to the lower levels. In another research, High-Level Cost Model [23] is used to automatic task level parallelization for multicore embedded systems. In this work, they use Augmented Hierarchical Task Graphs including several optimization techniques in

order to perform an automatic partitioning and mapping of software to heterogeneous hardware. Ceng et al. [24][25] present MAPS (MPSoC Application Programming Studio) which performs a semi-automatic task-level parallelization.

With regard to parallel programming interfaces, several kinds of computer program parallelization standards are available in the literature like OpenMP, MPI, and CUDA. The MPI usually works better in a distributed system with a cluster environment, where CUDA is more used for GPU computation. In the recent works, the ScalScheduling [26] shows an improvement of the traditional interactive application design which overcomes the data transfer rate as well as the task scheduling latency over the network. However, the OpenMP task parallelism technique is well-known for shared memory architecture in a multiprocessor system. In the OpenMP task parallelization technique, each task is created inside a parallel region with a thread that can split itself into a number of threads following the fork and join model.

OpenMP 4.5 and 5.0 introduces a tasking model that supports both task dependency and nested task parallelization. Recent work shows an extension to OpenMP 5.0 that improves the taskwait barrier and task dependencies from an inner task to outer task exposing more parallelism [27], but still, there are no significant changes in the improvement of task scheduling approaches. The OpenMP 3.0 [28] version has different loop scheduling techniques which can be implemented using #pragma omp parallel for schedule () [10]. The choice of scheduling types (e.g., static, dynamic) depends on the application type and their requirements. A proper selection of scheduling technique can result in substantial performance gains [29]. Considering this goal, an adaptive scheduler [30] is designed for an OpenMP compiler that helps to find the best scheduling policy reducing the number of threads at runtime.

In a recent study, Melani et al. propose a static scheduling approach that enables safety-critical OpenMP applications [11]. In their proposed method, they focus on deriving the optimal mapping of task-parts to threads using ILP formulation by introducing a static allocation model for tied and untied tasks. In addition, TG-PEDF [31] shows a new way of scheduling mixed-criticality tasks based on high- and low-criticality tasks grouping. It implements MILNP formulation to analyze the tasks grouping, and the evaluation of this strategy outperforms the mixed-criticality (MC) scheduling algorithm. In another

study, a new benchmark suite OpenTGB presents a way to collect real-time tasks and transform them into task graph model [32]. The OpenTGB makes the real-time tasks eligible for scheduling efficiently by defining a new response time bound.
In our work, we demonstrate a general framework for applications with varying requirements and modify the OpenMP static scheduling approach for deterministic task execution. This framework will help to understand the effect of the modified scheduling for different requirements of tasks over the existing approaches in embedded systems.

2.3 Approaches to monitoring

A monitoring approach involves observing the execution behavior and performance of an application to understand the insight of a software operation [33]. To monitor an embedded system, a component of hardware or software is separately integrated into the system. The attachment of the component is called a probe. A probe extracts all the information about internal operations. As shown in Figure 2.1, the probe provides the generated output from the system to compare with the expected output. This can be achieved using a hardware probe that monitors internal system signals or any software probe that performs some functionalities in order to calculate the expected output.

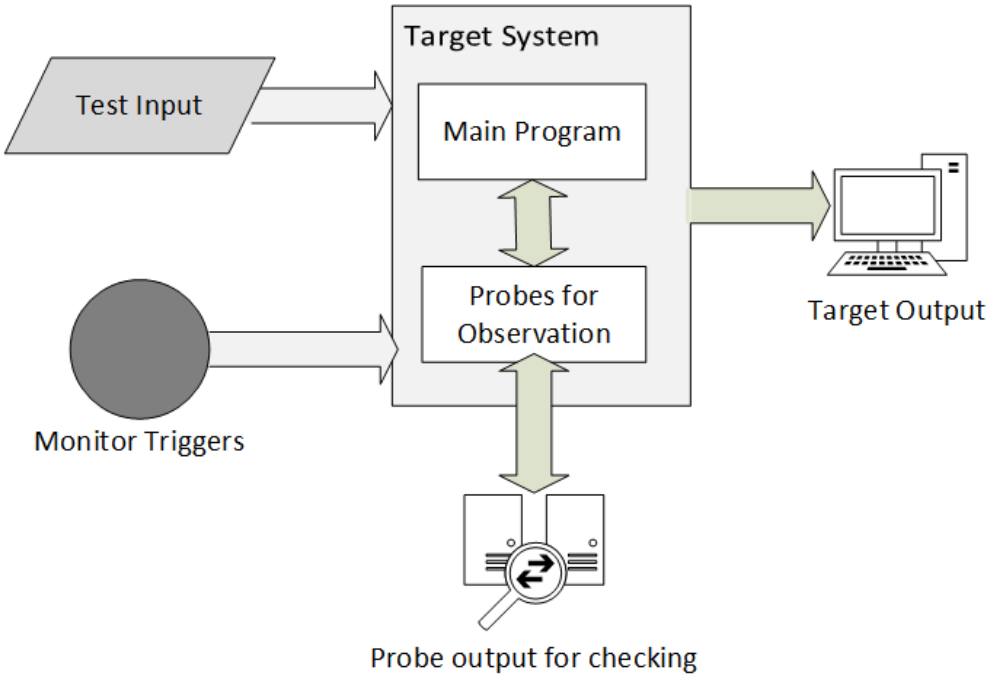


FIGURE 2.1: Output monitoring in embedded systems

A monitoring approach can provide a runtime verification that checks the system execution behavior and provides notification if any unexpected situation occurs at runtime. The execution behavior of the system is specified against the application's software requirements. Assuring the correct behavior and output of embedded systems is always a challenging job. In a traditional approach, a runtime verification can provide a solution in finding errors and the reasons behind those errors. A monitoring approach may vary in terms of the spheres of embedded systems.

2.3.1 Hardware Monitors

In modern embedded systems, the designers examine different kinds of metrics like cost, power, performance, resource utilization and so on. These parameters can be visualized and monitored through a hardware probe. The monitoring of these parameters can verify the actual output of the system.
One advantage of using hardware monitoring is that it reduces the possibility of intrusion caused by the internal or external execution of the monitoring code. However, it has a lot of limitations like the communication to different components may interrupt the actual execution and performance of tasks.

2.3.2 Software monitors

In the case of software-based monitoring, a software program is integrated into the operating system, or it can be executed via external hardware. The program code needs to be executed along with the main program which might cause a delay in the execution of the software code. The main advantage of a monitoring approach is that it can have access to a large amount of information of a complex embedded system [34].

2.3.3 Hybrid monitors

Another popular monitoring approach is hybrid monitoring that combines both the hardware and monitoring approaches. As a result, it can provide more support than others. At the same time, it alleviates the disadvantages of other approaches as it uses physical interfaces that have less interference with the system and captures the maximum information through software components. The hybrid approach may experience an overhead for executing additional software code which likely affects the behavior and

performance of embedded systems, but the consequences can be diminished associating the hardware monitoring.

2.4 Task scheduling approaches in multiprocessor systems

With the increasing availability of multiprocessors, many RTSs prefer the multiprocessor architecture for high computational tasks. Although the multiprocessor systems support parallel tasks execution, still, research is ongoing to find better strategies to schedule tasks efficiently. To schedule, a n number of tasks on a m number of processors, few scheduling algorithms show some advantages and disadvantages related to the deadline miss and overall performance. Moreover, the characteristics of multiprocessors are also important in tasks scheduling. Multiprocessors can be categorized as follows [2]:

- Type 1: each processor possesses similar computational rate and abilities,
- Type 2: each processor has a similar configuration of capabilities including varying speeds, and
- Type 3: each processor is separated from each other and heterogeneous.

One of the significant contributors to the multiprocessor scheduling problem is resource allocation. This problem states that the scheduling algorithm requires to map the appropriate task in the proper processor. Besides, it demands the timing guarantee for each task to ensure the stability of the system.

2.4.1 Scheduling approaches

In embedded multiprocessor systems, the static, dynamic, partitioned, and global [2] scheduling algorithms are frequently used to schedule the tasks. In addition, we see increasing use of rate monotonic scheduling approach for meeting the deadline of high priority tasks in embedded systems.

Static scheduling: In the static scheduling approach, the execution of tasks is controlled following a specific order that defines how the program execution threads will be running in the code at compile time. In any application, if it requires to control (e.g., lock, semaphores, joins, and sleeps) the threads to fulfill the requirements, the static scheduling approach is practiced generally. The partitioned scheduling approach eventually follows the static approach which is discussed in the later part.

Dynamic scheduling: The dynamic scheduling approach does reflect any control over threads. In this case, the thread scheduling is done by the operating systems based on any scheduling algorithm implemented in OS level. The order of threads execution depends on the integrated algorithm except the control is defined at compile time. However, this approach also establishes an order of execution, but the hardware performs this rather than the compiler. The dynamic scheduling shows faster task execution than the static scheduling as it has a free flyer without any intentional waits.

Partitioned scheduling: A partitioned scheduling algorithm allocates resources for an individual task to a particular processor [2]. Fig. 2.2 shows an example of partitioned scheduling each task τ_i is assigned on the individual processor. For example, τ_1, τ_2 and τ_3 are running separately in processor *one*, *two* and *three* respectively where processor *three* will remain idle later as no task is assigned in it. The partitioned scheduling algorithm has both pros and cons. The algorithm has the following advantages:

- It allows static task assignments.
- Partitioned scheduling supports most of the single processor scheduling approaches.
- If any high-criticality task runs on overrun mode, it affects only the task-associated processor where other processors remain unaffected.
- As tasks are not allowed to switch the processors, there is no migration cost.
- To design partitioned scheduling algorithms, each processor maintains a separate queue for handling tasks.

The principal limitations of partitioned scheduling algorithms are listed below:

- This algorithm cannot help to share all the available resources equally
- Since tasks avoid migration, the system may experience a low processor utilization
- Although any processor can become idle at runtime, it cannot be assigned for mapped tasks to other processors.

Global scheduling: Using a global scheduling algorithm, all the tasks can also be scheduled upon multiprocessors. The global queue cannot be applied to overcome the multiprocessor scheduling problem in safety-critical systems because it allows task migration among the available processors. The migration of any task usually adds extra

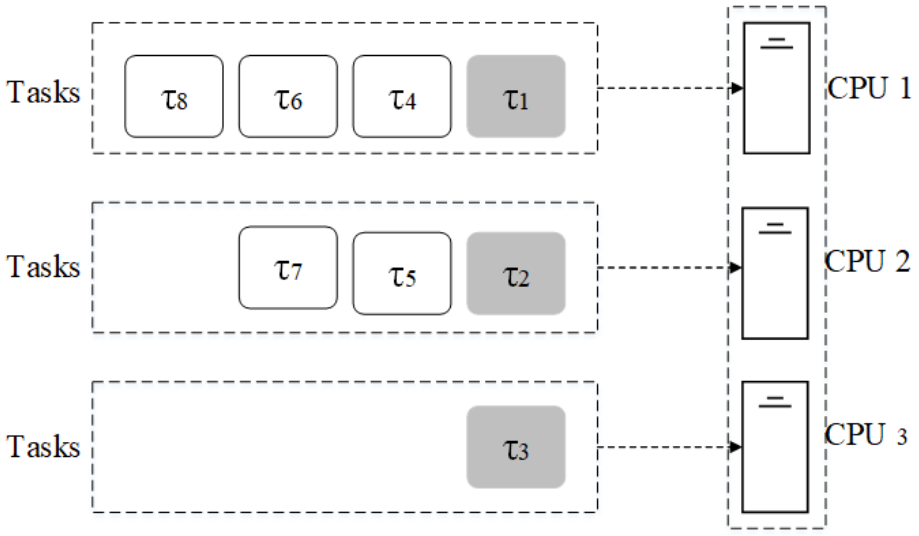


FIGURE 2.2: Partitioned Scheduling [2]

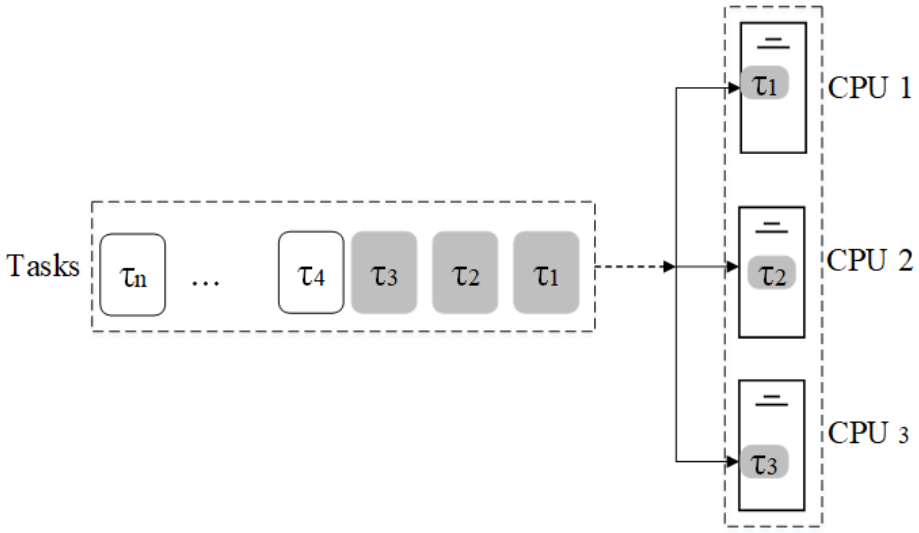


FIGURE 2.3: Global Scheduling [2]

overhead, and the system may show unpredictable behavior due to the dynamic changes in the task environment. Since the algorithm maintains a single global queue, the implementation and design become more complicated for different mixed-criticality applications [2]. Fig. 2.3 presents an example of global scheduling where τ_4 may be scheduled but it can never happen in partitioned scheduling approach. In comparison with the partitioned scheduling approach, the advantages of global scheduling are given below:

- All the immediate tasks are kept in a single global queue.
- Since task migration is allowed, the global scheduling ensures the maximum resource utilization.

- Most of the multiprocessor operating system now supports global scheduling.

On the other hand, some of the limitations of this approach are addressed below:

- Most of the uniprocessor scheduling algorithms cannot utilize the processors completely in multiprocessors.
- Task migration may lead to an unpredictable state for the safety-critical system.
- Due to task migration, various processors may cause varying computational rates which decrease the optimal performance.
- Tasks migration causes an extra overhead in the system because all the processors use shared memory for communicating among them.
- This approach may provide poor resource utilization for high-criticality tasks because of the prioritization of low-criticality tasks.

Rate monotonic: The Rate Monotonic scheduling approach assigns priorities to different tasks according to their time period. If the time period of a task is small, it will have the highest priority in case of execution. This approach follows the task preemption policy where a higher priority task can preempt the lower priority task from its execution. A given task is schedulable in an RM scheduling approach if it passes the schedulability test which is given in equation 2.1.

$$U(\tau) = \sum_{1}^{n} \frac{e_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.1}$$

Here, U(τ) represents the resource utilization, e_i is the computation time, T_i is the release period and n is the number of tasks.

2.4.2 Schedulability test

To schedule successfully any periodic task set on m processors can have utilization U(τ) no greater than $\frac{m^2}{2m-1}$ [35].

$$U(\tau) \leq \frac{m^2}{2m-1} \tag{2.2}$$

In a safety-critical system, the partitioned algorithm is used more than the global scheduling upon a multiprocessor platform. The reason behind the choice of partitioned scheduling is to use the processors better where global scheduling shows more unpredictability due to processor migration and unavailable cache information.

TABLE 2.1: A taskset with different requirements in a multiprocessor system

τ	Requirement	Deadline	ExecutionTime
τ_1	Low QoS	2	1
τ_2	High QoS	5	2.5
τ_2	High QoS	5	3

For example, Table 2.1 shows a set of tasks and these tasks have different requirements that need to be achieved. When scheduling these tasks in a 2-core system, the schedulability test using Equation 2.2 indicates that the tasks are not schedulable in static scheduling because process utilization is higher than the defined value. Therefore, not all tasks meet their deadlines.

CPU utilization calculation:

$$\Rightarrow \mathrm{LCM}(2,5,5) = 10$$
$$\Rightarrow \mathrm{U}(\tau) = \tfrac{1}{2} + \tfrac{2.5}{5} + \tfrac{3}{5} = 1.6$$
$$\Rightarrow \mathrm{U}(\tau) \nleq (\tfrac{\mathrm{m}^2}{2\mathrm{m}-1} = \tfrac{4}{3} = 1.3)$$

In uniprocessor and multiprocessor systems, once system utilization becomes higher than the optimal value, then the low-criticality or high-criticality tasks are not guaranteed to be scheduled to meet the deadlines.

Nevertheless, existing schedulability tests are not always enough to guarantee the timing behavior of a task if the other important factors like component's accuracy and reciprocal interference produced by the concurrent access to the shared memory resources are not examined correctly. Such factors can introduce lower data processing rate, variable delays, and jitter in the execution of tasks for which a system may lead to an unstable state with performance degradation [36]. Therefore, Yifan et al. [37] propose a general methodology to select the proper parameters for real-time controllers in resource-constrained systems. In this work, it shows that the selection of an appropriate period and deadline [38] can substantially improve the control performance in embedded systems. The experimental result in TrueTime [39] indicates the performance

improvement where the tasks are scheduled using EDF on a uniprocessor system.

2.4.3 Accuracy-related task scheduling

A number of task scheduling algorithms are available in embedded systems where the selection of an algorithm depends on the requirements of the systems. Most of the popular existing algorithms are familiar with the timing guarantee and efficient resource utilization of tasks. However, the correctness of output and QoS of a task are not discovered much compared to other task parameters. Existing embedded control systems have a tendency to tolerate a certain amount of inaccuracy in the task output. Following a similar concept, Buttazo et al. [40] propose an elastic task model where a task can intentionally change their execution rate for the different quality of services. Considering the current workload of a system, many QoS guided algorithms [41] are proposed that focuses on achieving high throughput and minimum execution time to keep the system stable.

To achieve the expected output, many research works focus on the selection of different parameters of a task which investigate the system performance and guarantee stability [42]. After that, different real-time task scheduling algorithms such as RM or EDF are applied for task schedulability analysis to ensure whether a task is able to complete its execution within the assigned deadline or not [43]. However, a system may experience an overload situation when it runs with limited resources than the required. As a result, many tasks may take a longer time to be executed than the usual time that is considered at the beginning of the design process. Thus, it shows the necessity to have a task schedulability test considering the worst-case execution time of all the tasks.

Mitra Nasri et al. [44] propose a scheduling algorithm depending on accuracy-constrained RTSs where two sources of data inaccuracy are considered like data noise and age of data. The defined accuracy of the work is regarded as a function of data noise which eventually indicates the QoS of a task output. In another work, Lin Huang et al. [45] propose a technique to improve the non-preemptive real-time scheduling allowing imprecision in the computation. As a result, a task that does not affect much for its timing

violation can be computed differently to achieve schedulability.

On another work, Bini and Cervin [46] show a way to calculate an approximate delay as a function of task periods and other obstructions that come during the performance optimization. As delays have an impact on the control performance, the proposed approach estimates an additional delay at design time so that it can significantly reduce the implementation-related performance degradation. It demonstrates that the proposed co-design method with fixed priority tasks provides a lower cost than previously offered period assignment schemes. However, in our work, we illustrate the DVFS technique for power-aware application where the system component's output accuracy need to be considered during task scheduling. Thus, we propose a calibration framework to monitor the accuracy of a system component.

2.5 Calibration for real-time embedded systems

In RTSs, every task is characterized by its deadline where each task is expected to perform a function producing a correct result within a specified amount of time. A hard RTS can lead to catastrophic failure if any task misses delivering the correct value at the right time. Although it is very important, most research works in RTSs avoid discussion on the correctness of values at different points in time. Measurement units or instruments can be integrated with RTSs to perform sensitive measurements where the measurement accuracy of a device is an essential factor for the precise result. Periodic inspections and calibrations of the measurement units validate the consistent measurement accuracy to ensure the safety of a system.

2.5.1 Uncertainty in measurements

A hard or weakly-hard RTS requires to respond correctly within a stringent timing deadline where the system is considered to have failed if it is unable to achieve the correct result within the allocated time. Therefore, hard real-time systems and cyber-physical systems (CPS) [47] may include measurement units to calculate the values of the critical elements. Some of the examples of critical measurements include the air pressure in an airplane, the voltage of a voltage divider, the viscosity of lubricant oil in engines, the rotation speed or torque of spinning wheels, heart rate, and electrocardiogram (ECG)

signal detectors in medicine [48]. However, the electrical measurement units may gradually lose accuracy for several measurements uncertainty factors like operational time, environment, electrical supply, process changes and more for which their responses naturally change over time. These factors have impacts in influencing parameters such as "offset" and "drift variations in gain" to cause uncertainties of measurements. Figure 2.4 shows the deviation line from the actual value measurement. These measurements uncertainty factors may unable to be avoided entirely, but calibration can amend it. An auto-calibration process can improve the accuracy of any measurement instrument significantly.

2.5.2 Purpose of calibration

Any system component or instrument calibration is mainly a way of monitoring systems to ensure that the input value in the measurements remains at its proper standing position. In terms of measurement technology, a calibration process [49] compares the measurement values delivered by a device under test with a standard calibration device of known high accuracy. Electronic devices measure different values such as current, voltage, resistance, and temperature along with the measuring reference values from standard equipment to compare deviations and perform calibration. By using the calibration process, the performance of any machine within a set range of specification can be easily verified. Typically, periodic measurement instrument calibration is vital to ensure accurate and repeatable readings. Calibration is not only necessary from a quality and consistency perspective but also a safety perspective. Therefore, first of all, monitoring correctness of measured values should be practiced on a regular basis in a safety-critical system, because any incorrect measurements could result in the loss of life or significant property and environmental damage. Furthermore, we need to ensure calibration happens when required, because an inappropriately calibrated measurement unit may potentially drive severe consequences and hazards in the environment due to not identifying anomalies in measurements.

2.5.3 Calibration steps

A calibration process can be generalized by dividing it into three steps. For instance: a) compare the result of a measurement instrument under test with an appropriate standard, b) documentation of the traceability provided by the national standard and c)

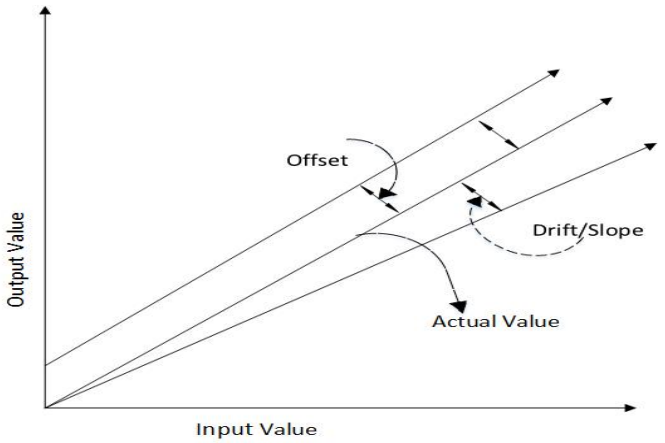


FIGURE 2.4: Drift and offset variation from actual measurement value

report of the deviation of the instrument from the standard. The first step of calibration includes the selection of standard [50] where it should have less than 1/4 of the measurement uncertainty of the device being calibrated. This also indicates that the standard equipment should be four times more accurate than the device under test. The correlation between the accuracy of the device under test (DUT) and the accuracy of the standard is known as test accuracy ratio (TAR). We can reduce the impact of the accuracy of the measurement on the overall calibration accuracy through ensuring a 4:1 TAR. In the second step, calibration essentially tests the device defined by accuracy and further provides specific correction values to use the device accurately in an application which is known as measurement traceability [51]. Thirdly, a report containing all the estimated variances and errors guides for the re-calibration or the adjustment process. .

2.5.4 Related works on calibration

The measurement accuracy and precision of any system have a significant impact on the reliable measurement result that is a prerequisite for any RTS. The decrease in measurement accuracy provides an implication of calibration for the measurement unit. For safety-critical systems, it is essential to run periodic or sporadic calibration for both new and old measurement instruments to ensure the correctness of measured values. To calibrate the measurement unit, the measured values are compared with the outcome of the reference or standard measurement unit as conformed with NIST.

In calibration techniques, many systems use external calibration standards to perform

the comparison with the measurement devices under test. For example, Guido Freckmann et al. [52] discuss the importance of system accuracy for Blood Glucose Monitoring Systems. Besides, they show the comparison of their measurement with the designated comparison method (manufacturers measurement procedure) to meet an acceptable accuracy. Although the manual calibration techniques are more seen in practice, many systems also use self-calibration techniques. In the self-calibration technique, the system does not require any external standards for the comparison.

For example, De Ma [53] presents a self-calibration technique for active vision system which directly uses the images from the environment rather than using a reference object. For other uses, particularly in many industrial control applications, the magnetic encoders (MEs) are practiced to measure speed and position where the signals of MEs receive error factors such as offsets, amplitude differences, and noise. Ha. Xuan Nguyen et al. [54] introduces an auto-calibration method using adaptive-bandwidth phase -locked loop (ABW-PLL) algorithms to reduce the errors and to determine the positions accurately. This method uses a low-pass filter (LPF) with an improved cutoff frequency that eliminates the noise of the MEs signals which helps to determine the calibration parameters easily.

However, according to NIST calibration handbook [55], the test accuracy ratio of the measurement unit under test and the standard measurement unit should be minimum 4:1. In many cases, it becomes a challenge to find out the standard calibration unit because of technological advancement. Shuyang Ling and Thomas Strohmer [56] show a different approach to achieve optimal performance of advanced high-performance sensors. In this approach, they combine self-calibration, compressive sensing, and biconvex optimization where the self-calibration process implements a smart algorithm for adjustments.

In real-time embedded systems, we do not observe the use of measurement standards that are used for maintaining calibration accuracy. However, many real-time and Internet of Things (IoT) applications require high precision accuracy and therefore require using calibration techniques for their measurement units.

Chapter 3

Design automation and QoS requirements preservation in multiprocessors

3.1 Introduction

Today's system application designers prefer higher levels of abstraction in the design process to avoid the complexity of writing embedded system applications for multiprocessors. Although existing parallel programming frameworks have made it easy to write program syntax for programmers, programming complexities exist from the high-level application design to implementation as the program grows. Moreover, these parallel programming frameworks are still not suitable for running tasks in hard or weakly-hard RTSs. The scheduling approaches that are used in these parallel programming frameworks cannot provide timing guarantee for high QoS requirement tasks. Therefore, we consider soft and weakly-hard RTSs for mapping high-level requirements and preserve these requirements using different approaches.

In both RTSs, we automate the design process by converting the AADL specification to programming C++ Code that holds all the periodic tasks. To convert into a C++ Code, we use an AADL to C++ converter (MDA tool) [9] as shown in Figure 3.2. After that, we divide our thesis work into two separate routes to handle the task requirements in soft RTSs and weakly-hard RTSs. For soft RTSs, we use the OpenMP parallel programming framework to map the high-level requirements to OpenMP semantics. We adopt a scheduling approach to meet the high QoS requirements in soft RTSs. We propose

thread to processor binding approach that can be adapted in the adaptation layer to preserve the QoS requirements based on timing.

On the contrary, we use the LITMUS-RT kernel for monitoring the task requirements in weakly-hard RTSs. The integrated partitioned RM scheduling approach is used to schedule the tasks. We propose a calibration framework to monitor the task output accuracy that leads to preserving the task requirements. The requirements are considered as the correctness of the output which is translated as accuracy. To understand the direction of our thesis work, we present a details workflow as shown in Figure 3.1.

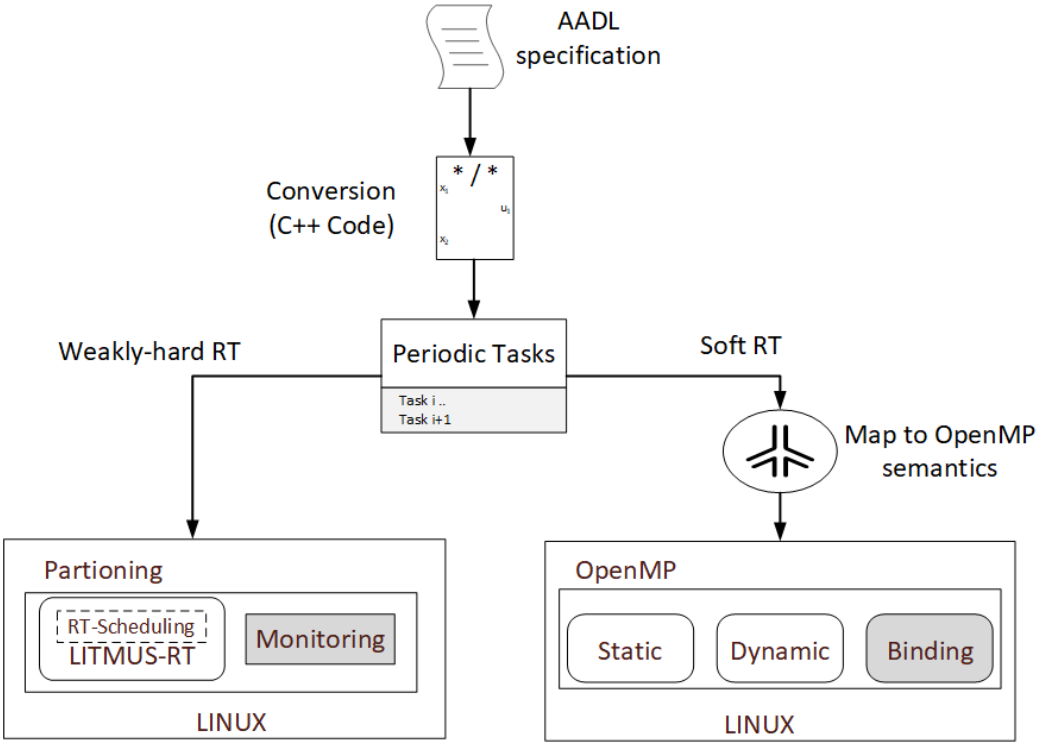


FIGURE 3.1: Details workflow of the proposed approaches

3.2 System model and assumptions

Let us assume we have a set of independent periodic tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ such that $n \in N^+$. Each task contains a number of jobs that are run for a certain number of times. A task is characterized as $\tau_i = (a_i, e_i, d_i, T_i, c_i) : (1 \leq i \leq n)$ where:

- a_i is the arrival time of task τ_i,
- e_i represents the execution time of task τ_i,
- T_i denotes the period of task τ_i,

- d_i is the execution time-bound where $d_i \leq T_i$ and
- c_i is the requirement of a task; $c_i \in$ {high QoS, low QoS}

Using the task properties described above, an embedded system application designer can connect other processing factors like memory, processor and communication protocol to map them in the parallel programming interface. To explain our design automation, we show an example of AADL specifications that define various kinds of software and hardware component types such as systems, processes, threads, processors, and buses. A designer will specify the properties of all tasks including their requirements in the description and thus it will be transformed into a suitable C++ Code format which can be used in any parallel programming interface. Figure 3.3 shows different high-level specifications of periodic tasks τ_i to τ_n in the AADL specified format.

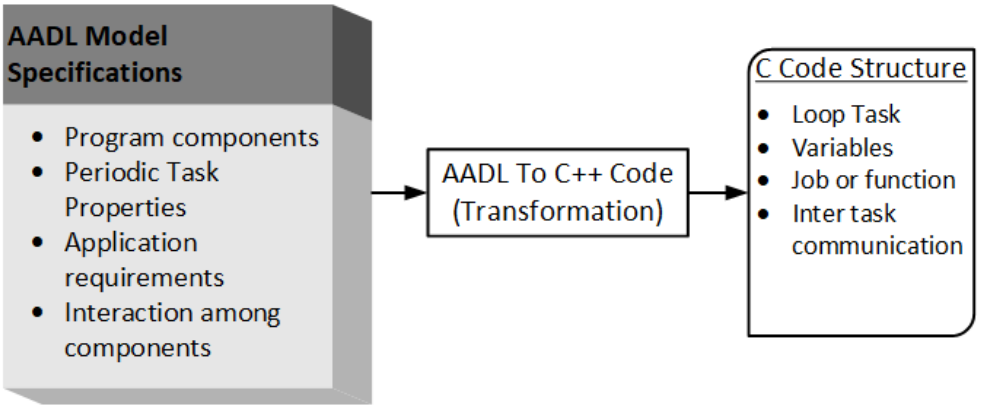


FIGURE 3.2: AADL model specification to C++ Code transformation

Therefore, we design a generic high-level layered infrastructure to automate the implementation of embedded system applications starting from capturing the tasks requirements and continuing to map the tasks in architecture resources. Figure 3.4 shows the layered architecture where the requirements of the periodic tasks are provided as an input in the application level and converted into a loop-based task construct. These converted tasks are considered either for weakly-hard RTSs or soft RTSs.

Weakly-hard real-time tasks: When the constructed tasks are considered for weakly-hard RTSs, our details assumptions for monitoring the task's output are defined in Section 3.4.2.

Soft real-time tasks: In the case of soft RTSs, the loop-based task construct is designed to map different parallel tasks to any parallel programming interface. Moreover, we add an adaptation layer on top of the parallel programming interface to handle varying requirements during the execution of different tasks. The adaptation layer is used to

```
system systemName
end systemName;

process process
end process;

process implementation process.imp
 subcomponents
   task τi: thread τi.imp;
   Source_Text: "prime_number code";
   task τi+1: thread τi+1.imp;
   ...
   task τn: thread τn.imp;
   Source_Text: "matrix_multiplication code";
 properties
   ArrivalTime: "ai","ai+1", …, "an";
   Dispatch_Protocol: Periodic;
   Execution_Time: "ei", "ei+1", …, "en";
   Execution_Time-bound: "di", "di+1", …, "dn";
   Task_Requirement: "Ci", "Ci+1", …, "Cn";
   Period: "Ti", "Ti+1", …, "Tn";
   Bus_Properties: Protocols: CSMA;
   Processor: "P1", "P2", …, "Pm";
   Job_Loop :τi[1, N],τi+1[1, N], …, τn[1, N];
end process.imp;
 … …
```

FIGURE 3.3: AADL model specifications

meet varying requirements that are either high QoS or low QoS. In our case, we adapt a task scheduling algorithm that binds a program execution thread (H) to a processor (P) to satisfy deterministic task execution for high QoS tasks and better throughput for low QoS tasks. Hence, the approach of binding application tasks onto different processors for a shared memory architecture intensifies the dynamic design process.

In the design automation process for mapping high-level requirements, the parallel loop-based task construct can be employed in any parallel programming framework, but OpenMP shows better and more distinct ways for task parallelization and scheduling. Therefore, we use OpenMP as a parallel programming interface to leverage the design process and implementation of embedded applications. We adapt the OpenMP static and dynamic task scheduling approaches in the adaptation layer to make it more suitable for meeting different requirements of tasks.

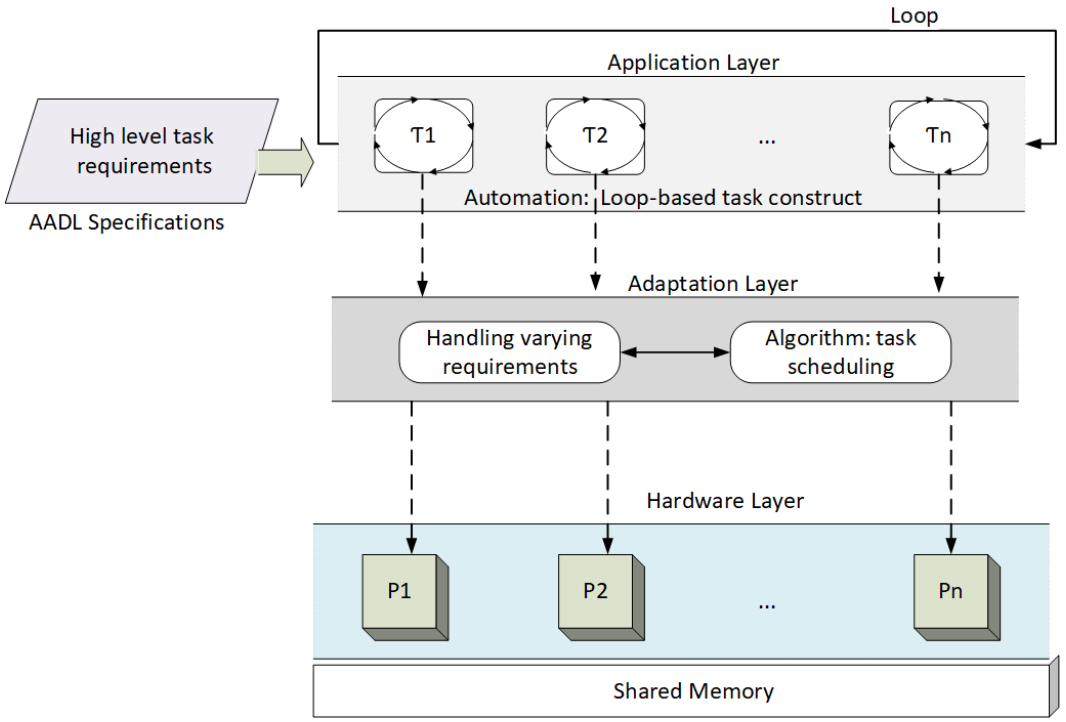


FIGURE 3.4: System model layered architecture

3.3 Design automation for mapping requirements

To assist in the automated synthesis of mapping high-level requirements into the tasks of a parallel programming framework and the proposed monitoring framework, we design a loop-based task construct for periodic tasks that can run in parallel with varying requirements.

3.3.1 Generation of a loop-based task construct from AADL

The design automation framework defines an AADL to C++ Code transformation module that converts the AADL specification to C++ Code relating different transforming rules which are commonly used in AADL model [57]. We divide this task construction process into two separate modules.

3.3.1.1 Identifying AADL components

This module mainly focuses on the process components, subcomponents, thread information, properties of tasks and different data components for generating C++ Code syntax. To convert the AADL components to a C++ Code, it follows a pattern of mapping that is defined in Table 3.1. The AADL components are mapped with each of the

task properties (e.g., e_i, d_i, T_i, P_i, H_i) and transformed into a loop-based task construct.

TABLE 3.1: AADL components to C++ Code mapping

AADL Component	**C++ Code generation**
System	Creates the application directory
Subcomponents	- Creates the AADL specified tasks inside a loop - Defines the number of tasks - Defines a function to execute each task
Thread	- Creates a thread to execute a particular task - Create a identifier for its port
Properties	- Maps all the properties of a task with the defined task in the subcomponents - Initializes all the variables according to a task timing parameters
Job_Loop	Creates an inner loop for executing jobs inside a task
Dispatch_Protocol	- Defines the scheduling policy from the dispatch protocol - Assigns a task to a processor
Task_Requirement	Initializes variables to define the requirements of tasks

```
for (int i = 1; i <= periodicTasks.tasks.length;
i++) {
    for (int j = 1; j <= N; j++) {
      //task τi
      //compute (ai,ei,di,ci,Ti,p1)
    }
    . . .
    for (int k = 1; k <= N; k++) {
      //task τn
      //compute (an,en,dn,cn,Tn,pm)
    }
}
```

FIGURE 3.5: A snippet of a generated C/C++ Code from AADL specifications

3.3.1.2 AADL specification to C++ code conversion

Figure 3.5 shows the generated C++ Code snippet where all the periodic tasks containing AADL specification are set inside an outer loop. These tasks are distributed under a `for loop` to run in parallel. In addition, a task which includes multiple jobs is constructed with another inner loop to control those jobs inside of it.

3.4 Proposed approach for QoS requirement preservation

The proposed approach considers two types of embedded systems for preserving QoS requirements. The requirements of tasks in soft RTSs vary on timing requirements. The high QoS requirement tasks require deterministic execution and other tasks may not need deterministic execution. On the other hand, the requirements of tasks in weakly-hard RTSs require a certain accuracy in its output along with timing guarantee. Thus, it necessitates a calibration framework that can ensure the expected accuracy by observing output continuously.

3.4.1 Preservation of QoS requirements in soft RTSs

To preserve the task requirements in soft RTSs, we provide a detailed description of our proposed framework as shown in Figure 3.6 to automate the application design process on a multiprocessor platform. The proposed framework uses a loop attribute extractor that helps to decide how many tasks and jobs you need to map in the parallel programming interface.

Loop attributes extraction: After the C++ Code conversion, the loop properties of the constructed tasks are extracted by executing a module called *loop attributes extractor*. This module extracts the number of tasks, the number of iterations for the available jobs and the requirements of the tasks to schedule them correctly on any targeted physical platform. After that, the designed framework transfers all these tasks information to the next module called the loop controller. The *loop controller* is responsible for evaluating all the tasks requirements. To do that a submodule named as *task analyzer* is integrated with it. Moreover, the *task analyzer* decides how a task should be separated from another task and how it will be executed in any Parallel Programming Interface like OpenMP to preserve its requirements.

OpenMP provides a loop scheduling library that splits the total number of iterations into different chunks of a defined size (K) and runs them parallelly in multiple processors to reduce the overall execution time [58]. We show how a task scheduling approach is selected based on the QoS requirements of tasks. Using the OpenMP schedule(sched_approach, chunk) method, the selected scheduling approach usually maps the whole iterations sequentially to multiple threads while the threads are run in parallel upon a multiprocessor system. The static scheduling approach assigns the iterations of a `for loop` evenly to the available threads. Alternatively, threads are assigned to the

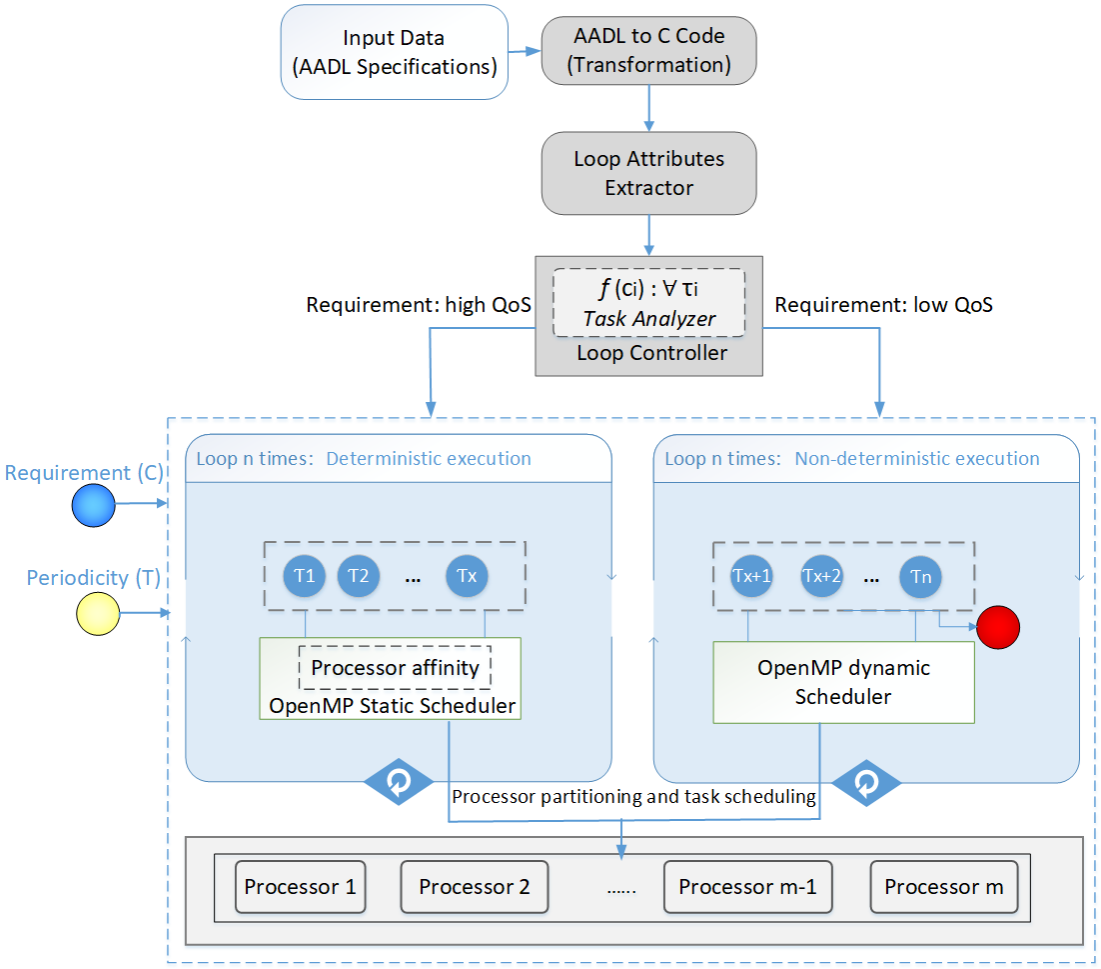


FIGURE 3.6: Proposed design automation framework for requirement-preserving loop-based task constructs

processors for every chunk size of iterations on a first come first served basis in the dynamic scheduling approach. Another approach exists which is called guided scheduling that works similar to the dynamic approach, but one significant difference is the chunk size that varies at runtime [12]. The user-defined chunk size guides to determine how the number of tasks should be distributed on multiple processors. The OpenMP has a `taskloop` construct that executes the iterations of one or more associated loops in parallel as OpenMP tasks. An example of the taskloop construct is shown in Listing 8 where the grain-size defines how many number of loops will be considered as a single task.

The proposed framework considers the OpenMP task scheduling techniques to schedule all the tasks with varying requirements. In order to schedule tasks, the framework starts to evaluate the requirements of tasks and select the scheduling policy to execute them properly.

```
#define sampleSize 10;
#pragma omp parallel
#pragma omp single
#pragma omp taskloop grainsize(1)
  for (i=0; i<sampleSize; i++) {
    /*execute_task();*/
  }
```

LISTING 3.1: An example of OpenMP `taskloop`

3.4.1.1 QoS requirements evaluation

To separate each task, the *loop controller* divides the taskset τ into two sets as τ_{high} containing tasks that have deterministic timing bound, and τ_{low} containing tasks that have no deterministic execution requirements. Equation 3.1 separates each of the task through a function $f(c_i)$ after evaluating the QoS requirement.

$$\forall\, \tau_i;\ f(c_i) = \begin{cases} \tau_{high} \leftarrow \tau_{high} \cup \tau_i & :\ c_i = \text{high QoS} \\ \tau_{low} \leftarrow \tau_{low} \cup \tau_i & :\ \text{otherwise} \end{cases} \tag{3.1}$$

3.4.1.2 Proposed thread to processor binding approach

In addition, we examine the feasibility of different scheduling algorithms to meet the requirements. In our approach, we analyze the deterministic execution behavior of OpenMP task scheduling policy to schedule all the tasks satisfying its requirements. Determinism is a property that describes how consistently a system responds to execute the tasks. The deterministic behavior of a multithreaded program can be affected by the scheduling policy, memory and system configurations [59]. After analyzing the deterministic execution behavior of OpenMP scheduling approaches, we propose a thread to processor binding approach which performs better than existing approaches. It binds each program execution thread $\mathrm{H_j}$ to a processor $\mathrm{P_l}$ that executes a task τ_i at a particular time $\mathrm{t_k}$ where $(\mathrm{j,k,l}) \in \mathrm{N^+}$. Figure 3.7 shows the details of the thread to processor binding approach.

3.4.1.3 Algorithm for requirements adaptation

Algorithm 1 presents the pseudocode of our proposed parallel loop-based task construct approach for a workload with varying requirements of tasks. In the static task scheduling approach, the proposed thread to processor binding technique is enabled for high QoS

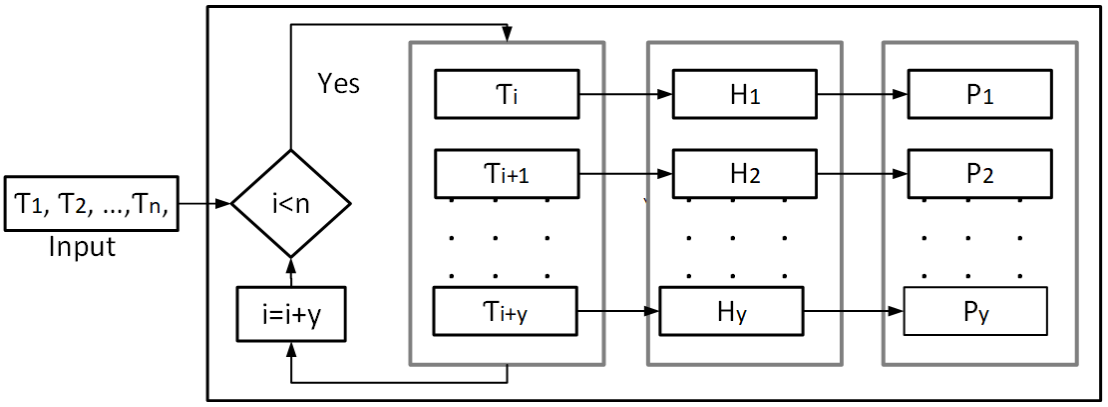


FIGURE 3.7: Thread to processor binding approach

Algorithm 1 Algorithm for adapting timing requirements

procedure MODIFIED–TASK–SCHEDULE($\tau[]$)
  $P \leftarrow \{P_1, P_2, \ldots, P_{m-1}, P_m\}$; $m \in N^+$
  $H \leftarrow \{H_1, H_2, \ldots, H_{y-1}, H_y\}$; $y \in N^+$
  HT[] $\leftarrow$ mappingTable(m, τ)
  / n is number of available tasks*/*
  / modified static approach for binding processor*/*
  #pragma omp parallel for schedule(static)
  for all task $\tau_i = \tau_i : (1 \leq i \leq n) \in \tau_{high}$ **do**
    / hash function to select a processor */*
    $hx = hx(\tau_i) = (i \bmod m) + 1$
    if HT[hx] is empty **then**
      HT[hx] $\leftarrow$ create new list
      $d_i \leftarrow$ check execution time-bound, τ_i
      $A_i \leftarrow$ CPU_AFFINITY(P_i)
      HT[hx] $\leftarrow$ add τ_i to HT[hx]
      executeTask(τ_k, d_i, A_i)
    else
      HT[hx] $\leftarrow$ add τ_i to queue
    end if
    $i \leftarrow i + 1$
  end for
  #pragma omp parallel for schedule(dynamic)
  for all task $\tau_k = \tau_k : (1 \leq k \leq n) \in \tau_{low}$ **do**
    $d_k \leftarrow$ check execution time-bound, τ_k
    */*based on the processor availability*/*
    $A_k \leftarrow$ Get an available CPU, P_k
    executeTask(τ_k, d_k, A_k)
    k++
  end for
end procedure

tasks to meet the respective execution time-bound of each task. We define this modified task scheduling approach as "Binding" scheduling. Alternatively, all the low QoS tasks τ_{low} are run in parallel using the dynamic scheduling approach that maps each task to a thread and assigns the thread to a processor based on the availability. During the high QoS tasks scheduling, a hash table is maintained to automate the task assignment based on the hash value of function $hx(\tau_i)$ as shown in Equation 3.2. It selects an available execution thread for a task τ_i and ties it with a processor. Moreover, we check the time-bound of each task before its execution. Then, we update the hash table (HT[]) to manage different operations of task scheduling. Once the task scheduling is completed, the *executeTask ()* method executes the individual task.

$$hx(\tau_i) = (i \bmod m) + 1 \tag{3.2}$$

To bind a thread to a specific CPU or processor, we use CPU_AFFINITY(P_i) method that includes a variable called GOMP_CPU_AFFINITY. The variable determines the sequences of threads to be matched with the processors. To understand the sequence of assignment, we consider an example GOMP_CPU_AFFINITY="1 2 3 4" that binds the thread H_1 to processor P_1, then the thread H_2 to processor P_2 and so on.

3.4.1.4 Adaptation constraints

To schedule all the tasks using the proposed approach, we define the following constraints:

- Execution of a task: At any point in tasks scheduling, a task should not execute more than one threads. Moreover, a task that requires high QoS can only be interrupted only at task scheduling point which indicates the completion status of a task. In contrast, a task that requires low QoS can be interrupted at any time to meet the requirement of higher priority tasks (high QoS).

- Thread binding for tasks with high QoS requirement: In our proposed approach, we enforce tasks that require high QoS to be executed on the same thread. On the other hand, a task with low QoS requirement can migrate to any threads based on the availability of the threads and processors.

For a given set of tasks τ and the available number of threads H, we present the task scheduling as a y-dimension vector of functions S = $(s_1, , s_y)$ where each function $s_u(t) =$

v such that $v > 0$ and $\forall(t, u, v) \in N$. This implies that the thread $H_u : u \in [1, y]$ is executing task τ_i at time t. Adding a scheduling constraint as shown in Equation 3.3, we ensure that no task will be executed more than one using two different threads simultaneously. Besides, we show that no thread will remain idle if the other tasks are ready for execution.

$$s_u(t) \neq s'_u(t) : \forall(u, u') \in [1, y];\ \forall \tau_i \in \tau \text{ with } t > 0 \tag{3.3}$$

Moreover, to handle the task preemption [60], we make sure that tasks that require high QoS should not be preempted once they start executing. Tasks that require low QoS are preemptable when required. Equation 3.4 shows the constraint for high QoS tasks where the constraint states that once a thread H_u begins to execute a task, it remains busy until that task completes its execution.

$$s_u(t) = (0, d_i) \text{ only if } \tau_i \in \tau_{high}(t) \tag{3.4}$$

The motivation of the proposed thread to processor binding scheduling approach for high QoS tasks is that they are prone to miss the execution time-bound in the conventional dynamic and static scheduling approaches due to thread synchronization overhead and data locality in caches. Finally, the framework emphasizes on the design automation from high-level requirements of tasks to low-level implementation in soft RTSs.

3.4.1.5 Discussion

Apart from static and dynamic allocation strategies, OpenMP allows another loop scheduling method called the guided scheduling approach which performs well in terms of balancing the workloads among processors. This approach starts with a relatively large chunk size and decreases to a predefined minimum size. The analysis of the chunk size at runtime and the adjustment of assigning loops to the thread make the guided scheduling approach weak taking a longer time. Moreover, the overall execution time extends longer due to the number of reduced chunk size at runtime. Therefore, although the guided scheme provides better load balancing among the iterations, it performs poorly in terms of execution time. Our loop-based task construct with modified static scheduler ensures more reliability as it binds high QoS tasks to processors at runtime. Such allocation allows us to attain more predictability to meet the respective time-bound of each task.

3.4.2 Preservation of QoS requirements in weakly-hard RTSs

The weakly-hard RTSs execute a number of tasks on different components (e.g., hardware and software) where each task is required to complete its operation within a stringent timing deadline utilizing limited computing resources. In some cases, the system may tolerate a specific amount of delay that is defined earlier. However, in any dynamic environment that changes frequently, a system including integrated components requires robustness in generating the correct output. The robustness requires a system or component to operate correctly in the presence of invalid input or stressful situation [61]. Moreover, the output of a task requires to represent a good QoS of the system or components. Therefore, we present a monitoring-based calibration framework that monitors the task output accuracy to preserve the task requirements. To understand the calibration workflow we show a state machine diagram that is shown in Figure 3.8. The design diagram explains how the monitoring module helps to calibrate a system. It also illustrates the transition of a safe state from different possible states. For example, if the monitoring approach finds an unexpected issue for component's accuracy, it directs to calibrate the system. After the successful calibration, the system continues to execute the tasks again. Otherwise, it moves to a secure state for safety.

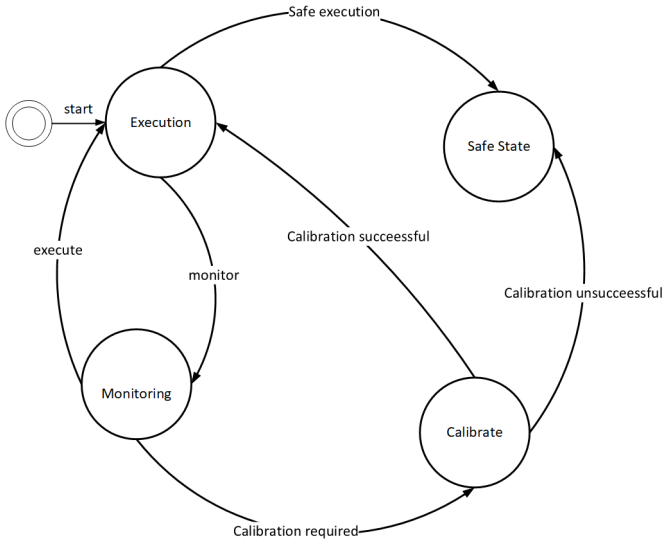


FIGURE 3.8: State machine model for calibration approach

To understand the accuracy of the task output, we define some new terms that are related to the system components. We consider an weakly-hard RTS with a set of n periodic tasks $\Gamma = \tau_1, \tau_2, \ldots, \tau_n$ running on a uni-processor using partitioning task scheduling algorithm where each task is defined as $\tau_i = (e_i, d_i, T_i, \pi_i, c_i, q_i)$; $0 < i \leq n \in \mathbb{N}$. In

addition, we assume each task is associated with a hardware and a software component that support in producing the final output. A component is defined by $M_i = (a_i^{acc}, a_i^{cur})$, where a_i^{acc} is the acceptable accuracy in the system which is known a priori and a_i^{cur} is the current working accuracy.

To analyze the impact of accuracy change of M_i on task τ_i, we introduce a new accuracy factor called error coefficient $\Phi_i(\%)$. We define the error coefficient ($\Phi_i\%$) as the difference between the acceptable accuracy specified by the user and the current accuracy of a component. For example, any component with acceptable output accuracy $x \pm 0.2\%$ and changed accuracy $x \pm 0.5\%$, the error coefficient Φ_i is $\pm 0.3\%$. Hence, in the new definition, $\tau_i = (e_i, d_i, T_i, \pi_i, c_i, \Phi_i, q_i, M_i)$, where

- π_i is the priority of a task (a lower numeric priority value corresponds to a higher priority),
- c_i indicates the requirement of a task, i.e., high QoS or low QoS. Alternatively, we can say the criticality level of a task.
- $\Phi_i(\%)$ is the error coefficient. The Φ_i becomes zero if M_i maintains the acceptable accuracy range.
- $q_i(\%)$ is the acceptable level [Min, Max] of QoS.

In our system model, we assume that each component's output trace is logged in a database where the initial traces of a component are considered as a calibration standard to use later for the calibration. We also assume that the manufacturers have correctly calibrated the system components. Thus, Figure 3.9 shows the framework for calibration of system components where the percentage of error in the output is minimized through a correction module.

3.4.2.1 Necessity of calibration: A rational example

Let us assume, we have three periodic tasks τ_1, τ_2, and τ_3 in a weakly-hard RTS. In addition, we consider each task as an implicit periodic deadline task as the relative deadline is equal to the period which is defined in Table 3.2.

To check the schedulability for the defined task set, we examine the Rate Monotonic (RM) task scheduling algorithm [62] throughout this work. The total CPU utilization for defined three tasks is 0.67 that ensures the successful task schedulability as the RM sufficient condition for CPU utilization of three tasks is 0.78.

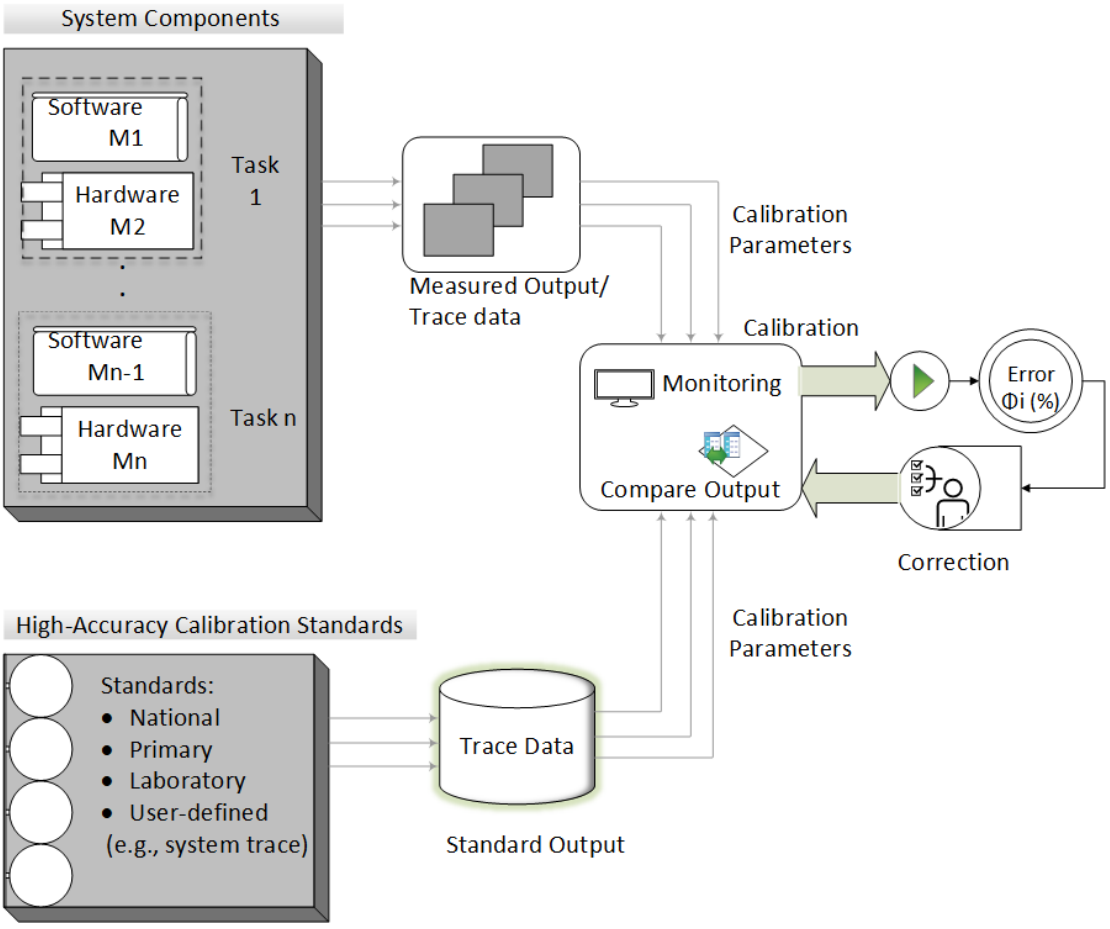


FIGURE 3.9: A calibration framework

TABLE 3.2: A set of three tasks in an weakly-hard RTS

Task	e_i (ms)	d_i (ms)	T_i (ms)	Priority, π_i	Requirement, c_i	$q_i(\%)$
τ_1	2	10	10	1	High	[100, 100]
τ_2	3	15	15	2	High	[100, 100]
τ_3	4	15	15	3	Low	[90, 100]

However, in any elastic task scheduling model, a periodic task can intentionally modify its execution time and period in different situations or mode changes [63] such as energy saving mode. To understand the applicability, we consider the DVFS technique that is used to minimize the power consumption of a high clock frequency system. Referring to the power dissipation (p) of a capacitor $\mathbf{C}'$, Equation 3.5 illustrates the details characteristics where V is the supply voltage for which the capacitor is charged to, f' is the frequency that the voltage is switched across the capacitor [64].

$$p \propto \frac{1}{2}\mathbf{C}' \times V^2 \times f' \tag{3.5}$$

In complementary metal oxide semiconductor technology, the change in a processor frequency for a supply voltage expands linearly. Equation 3.6 shows the processor frequency mapping [64] to the supply voltage where V_{th} is the threshold voltage, and δ is a constant.

$$f' = \delta.\frac{(V - V_{th})^2}{V} \tag{3.6}$$

To visualize the relationship between frequency and execution time, let us assume a 16-bit timer/counter where the clock frequency is 4.26MHz with a maximum of a 19V supply voltage. We also assume the threshold voltage defined by the manufacturer company is 10V which is the minimum required voltage to keep the system running. Thus, the clock counter gets overflow after every $2^{16} = 65536$ ticks or 15.384ms. For any individual task τ_i, the execution time e_i can be derived from Equation 3.7 as we know the required number of ticks K_i for a task is equal to the clock frequency multiplied by the execution time of the task τ_i.

$$e_i = \frac{K_i}{f_i'} \tag{3.7}$$

Due to the voltage scale down by $7.9\%(17.5\,V)$, the additional execution time $X_i^{V(v)}$ for each task τ_i is defined using Equation 3.8 where $10 < v \leq 19$. Thus, we estimate new execution time $e_1^{new}(2.65)$, $e_2^{new}(3.98)$, and $e_3^{new}(5.31)$ for the available tasks assuming $\delta = 1$ for simplicity.

$$X_i^{V(v)} = \frac{K_i}{\delta.\frac{(V_{new} - V_{th})^2}{V_{new}}} - \frac{K_i}{\delta.\frac{(V - V_{th})^2}{V}} \tag{3.8}$$

The updated total CPU utilization ($0.88 \nleq 0.78$) exceeds the sufficient condition. Hence, we check the necessary condition of RM scheduling. Equation 3.9 states a necessary condition for the schedulability test considering worst-case timing interference (I_i^e) from higher priority tasks where the worst-case response time $r_i \leq d_i$ [65]. The timing interference from a task j on task i is defined by I_i^e, where $1 \leq j \leq i$.

$$r_i = e_i + I_i^e \leq d_i \tag{3.9}$$

$$r_i = e_i + \sum_{\forall\, j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil * e_j \leq d_i \tag{3.10}$$

An iterative technique is used to solve Equation 3.10 where the iteration starts with $r_i^0 = 0$ and terminates when the $\mathrm{r_i^{(n+1)}} = \mathrm{r_i^n}$. The iteration is guaranteed to converge

if the CPU utilization is less than one. As an example, the schedulability test for task τ_3, the value of $r_3^3(14.59) < 15$ ensures the task schedulability. From the above

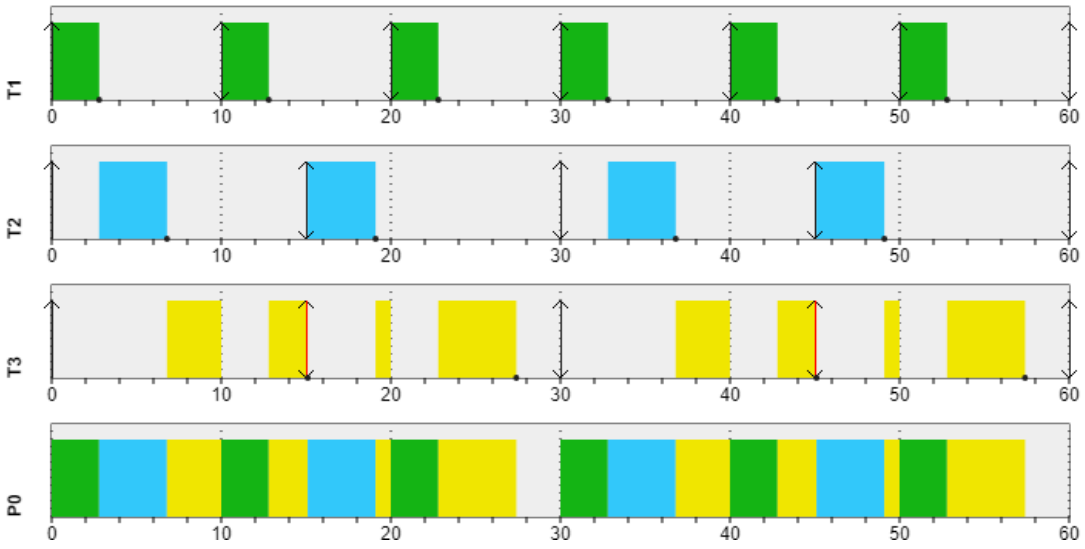


FIGURE 3.10: An example of a deadline miss due to inaccurate voltage output

theoretical analysis, we observe that the task set is only schedulable when the supply voltage remains in between $19\,V$ to $17.5\,V$. However, the task set is not schedulable if the voltage divider [15] component produces an incorrect output voltage for which the output voltage goes out from the tolerable bound. Figure 3.10 shows that the task τ_3 misses its deadline when the hardware component produces $17\,V$ instead of expected output $17.5\,V$. Thus, the accuracy of producing correct result motivates to integrate it in task schedulability test along with the theoretical timing guarantee.

3.4.2.2 Finding execution time delay for output inaccuracy

The objective of this work is summarized as follows: For any task τ_i in a given task set Γ, determine the execution time delay due to the change in the output accuracy of a hardware component. In a system with the supply voltage V, the threshold voltage V_{th}, and the required number of clock ticks K_i for task τ_i with execution time e_i, the execution time delay (Δ_i) is a function of error coefficient Φ_i.

$$\Delta_i = f(\Phi_i) = \begin{cases} 0 & : \text{for } \Phi_i = 0 \\ e_i - \frac{K_i}{\frac{((V \pm \Phi_i) - V_{th})^2}{(V \pm \Phi_i)}} & : \text{otherwise} \end{cases}$$

3.4.2.3 Proposed calibration framework

To perform the calibration, we follow several working steps that are defined in our framework. Thus, the proposed framework as shown in Figure 3.11 is composed of several modules: task admission controller, task scheduler, task accuracy monitor for analyzing QoS system performance as well as error correction.

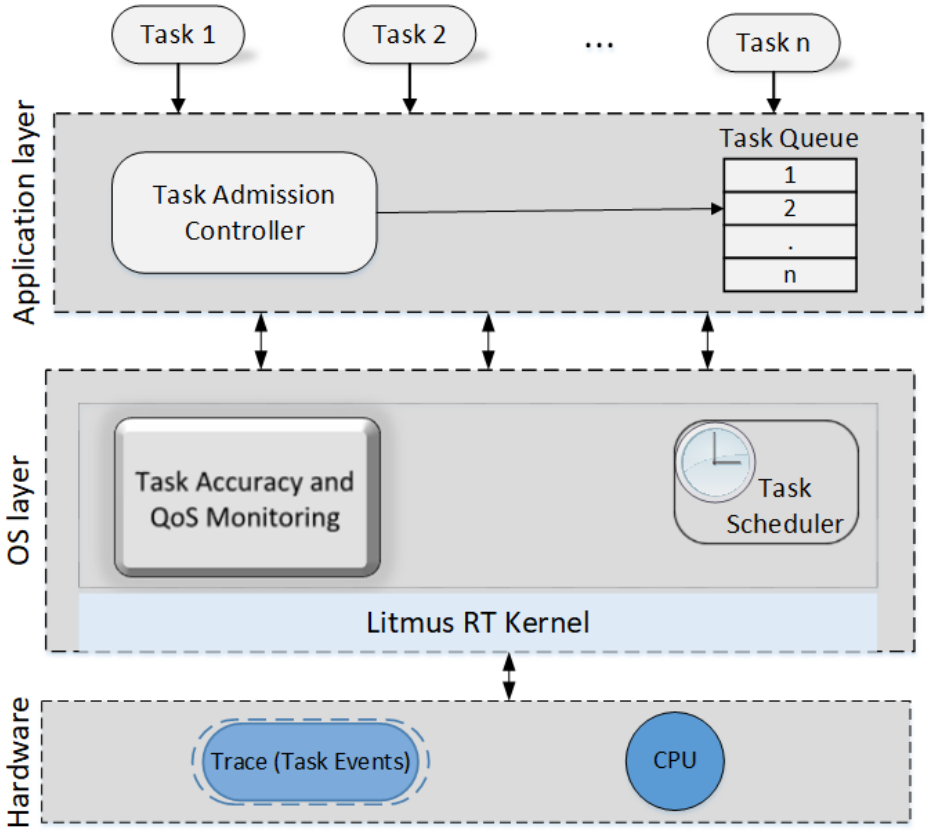


FIGURE 3.11: A calibration framework for monitoring accuracy

Admission controller: The task admission controller allows to change a task property based on the system requirements. This module can accept and reject a task as well as control the task arrival and release time to meet the system goals.

Task scheduler: In this module, a task scheduling algorithm is selected to perform some routine tasks at a convenient time. In our proposed framework, we only analyze the RM scheduling algorithm to understand the effects of component accuracy. The newly introduced error coefficient due to change of component's output accuracy shows the correctness of the result that affects the task execution time and deadline. Thus, Equation 3.11 and 3.12 show the condition to select a new e_i^{new} and d_i^{new} so that a task having current CPU utilization (U_i is equal to $\frac{e_i}{T_i}$) remains schedulable maintaining

the system-defined maximum CPU utilization (U_{max}). We allow a task to extend its execution time to a certain limit based on the available budget of the resources. Similarly, the new deadline can be selected between the minimum and maximum value that are defined in Equation 3.12. One way to set the deadline of a task τ_i is to take the minimum in between the current deadline and the remaining time after executing the last task. However, the deadline d_i should never exceed the period T_i.

$$e_i^{new} = e_i + \left(\frac{[U_{max} - \sum_{i=1}^{n} U_i]}{n}\right) * d_i \tag{3.11}$$

$$d_i^{new} = \begin{cases} min : min[d_i, d_n - e_n] \\ max : T_i \end{cases} \tag{3.12}$$

Lemma 3.1. *For a given set of n periodic tasks with $e_i^{new} \leq d_i$ and $d_i^{new} \leq T_i$, the task set Γ becomes schedulable using RM algorithm if the following conditions are satisfied.*

$$r_i^{new} = (r_i + \Delta_i) = (e_i^{new} + I_i^e + \Delta_i) \leq d_i \tag{3.13}$$

$$d_i^{new} + \Delta_i + I_i^e \leq T_i \tag{3.14}$$

A task set (Γ) becomes RM schedulable if and only if the deadline of a task is not less than the total response time (r_i) that combines the worst-case execution time(e_i) and the timing interference (I_i^e) from higher priority tasks [66]. For task τ_i, the maximum response time is a time-bound function, is given by $tbf(r_i)$. Thus, the necessary condition for τ_i to be schedulable is $tbf(r_i) \leq d_i$. Following the same condition, the execution time delay (Δ_i) provides the new response time (r_i^{new}) for which Equation 3.15 satisfies the schedulability condition as well.

$$\forall \tau_i \in \Gamma : r_i^{new} \textbf{ or } (tbf(r_i) + \Delta_i) \leq r_i \tag{3.15}$$

Monitoring component accuracy and task output: This monitoring module is run in the background of a system to log the traces during the task execution. The initially recorded traces of the tasks are used as the calibration standard to compare with the current output. The QoS of a task and the output accuracy of a component are monitored by analyzing the collected traces. Once we find the execution time delay

due to inaccuracy which is analyzed from the collected traces, the proposed task schedulability test notifies about the component's output accuracy change. It also guides when a system component requires to calibrate. A set of calibration standards can be placed to monitor the change of component accuracy. In our thesis, we consider the following pattern matching of trace data as a standard:

Pattern matching from trace data: To implement a software-based auto calibration, the framework analyzes the trace data of all the task events. The primary purpose behind this trace analysis is to find a natural functioning pattern with associated values so that the generated output can be monitored from the previous traces of the system. In our experiment, we apply different voltage levels to the system and trace the related frequencies against each voltage level. Initially, we assume that the hardware equipment is calibrated correctly. Moreover, we determine a pattern $< V_i, f_i, C_i(\tau_i), q_i, a_i^{curr} >$ which is usually maintained during the initial runtime of the system. Afterward, to find out the appropriate crisp output as a standard for a given crisp input (X), we use the fuzzy logic control where the degree of membership in each fuzzy input set is calculated using linear interpolation. The fuzzy set theory defines the similarity of finding a pattern to a previously identified set of patterns (Y). The valid matched of the trace pattern is compared to examine the amount of deviation from the current output pattern. Finally, Algorithm 3 presents an overview of our proposed workflow.

$$
\begin{aligned}
\mathrm{X} &= \{< V_i, f_i, e_i(\tau_i), q_i, a_i^{curr} >\} \\
\mathrm{Y} &= \{< V_1, f_1, e_1(\tau_1), q_1, a_1^{curr} >, \\
&\quad < V_2, f_2, e_2(\tau_2), q_2, a_2^{curr} >, \\
&\quad \vdots \qquad\quad \vdots \qquad\quad \vdots \\
&\quad < V_n, f_n, e_n(\tau_n), q_n, a_3^{curr} >\}
\end{aligned}
$$

Algorithm 3 defines a `compute-error-coefficient()` function that takes the trace data as an input for each system component and estimate the error-coefficient $(\Phi_i\%)$ comparing with the initial trace data. Moreover, after retrieving the Φ_i, it checks the task schedulability using `check-task-schedulability()`. This method returns the response time which we compare with the deadline of a task τ_i. Similarly, the

quality of service q_i of a task is computed using a user-defined function called `get-task-QoS(trace [])` that matches with system defined bound. Finally, the algorithm checks the schedulability and QoS to take action for calibration.

Algorithm 2 Algorithm for a component calibration

```
1: struct CPU-Configuration *par;
2: procedure ACCURACY_AWARE_SCHEDULING (M_i, Γ, U_max)
3:     U_curr ← compute current CPU utilization
4:     // configure CPU & voltage scaling parameters
5:     CPU_SET (par→voltage, par→scheduler);
6:     // initiate trace logging
7:     trace [] ← [sched_trace(), ft_trace(), litmus_log()]
8:     if (U_curr < U_max) then
9:         for each task τ_i : ∀(1 ≤ i ≤ n) do
10:            (e_i^new, d_i^new) ← assign new timing property
11:            // task output monitoring
12:            Φ_i ← compute_error_coefficient(M_i, trace[])
13:            r_i ← check_task_schedulability();
14:            q_i ← get_task_QoS(trace[]);
15:            if ∀(r_i ≤ d_i)&&(q_i() defined_range(τ_i)) then
16:                printf("task is schedulable");
17:            else
18:                printf("task is not schedulable");
19:            end if
20:        end for
21:    end if
22: end procedure
```

However, in our work, the baseline of a component accuracy is determined from the voltage supply of the DVFS algorithm implemented in the LITMUS-RT kernel. A component is required to calibrate if there is a deviation of the voltage supply value than the expected value. In our experiment, initially, we assume that the component requires no calibration and therefore, the configured voltage value and the supply voltage are accurate. After that when we reduce the expected voltage by the different percentage of voltage (e.g., 5%, 10%, and 15%), we assume there are inaccuracies in the actual voltage supply with a margin of error. We use the synthetic data of accuracy changes in the experiment to demonstrate the importance of a component's calibration in RTSs.

On the other hand, we determine the QoS of a task from its throughput. The throughput is calculated based on the amount of data being processed, which we have captured using the LITMUS-RT tracing mechanism. In our work, we also assume that the acceptable value of throughput for a particular QoS will be determined by an engineer. Therefore, a user-defined function in Algorithm 3 is provisioned to select the required parameters.

Error correction: If the task set is not schedulable, our proposed framework tries to schedule the task by adjusting the timing properties of a task. After that, if the task set remains not schedulable, it notifies the system to calibrate its components. The calibration of components can be divided into two, such as software and hardware. If the output inaccuracy of a task happens for a software component, the software rejuvenation can be employed proactively to correct the output error. However, the hardware component needs to be calibrated with a standard when the software component acts fine.

3.4.2.4 The working principle of a software-based calibration

To understand how a software-based calibration can be implemented, we discuss one of our preliminary works of a resistive voltage divider. In this study, we propose a software-based monitoring system for not only detecting anomalies of measured values but also the calibration process of measurement units under test to ensure their correctness. This approach is particularly targeted for real-time embedded systems where incorrect values have major consequences. The proposed approach will provide an automatic indication when the measurement component needs calibration to correct itself before any anomalies of measurements have happened. One of the key characteristics of our approach is that we check the TAR sporadically ratio with the standard. We propose an algorithm that checks the current TAR and notifies the system when the minimum ratio is not maintained.

In this experiment, we consider a voltage divider as a walkthrough example that converts from a large voltage to a small voltage through multiple stages where the voltage measurement of each stage requires a certain accuracy. We present methods to find ranges of measurement for different components of a voltage divider which eventually provides a signal for auto-calibration or correction of the voltage divider when its values exceed the range. Usually, resistive voltage dividers use various switching techniques during auto calibration and measure voltage transfer ratio to calibrate the voltage at each stage [67]. Existing hardware specific calibration techniques provide limited opportunity to monitor the output of all internal stages [68]. However, through this software-based monitoring approach, we can guard the output voltage of each stage by determining a specific voltage bound to provide feedback when any unusual events take place.

a) Assumptions for software-based calibration:

In the calibration process of a weakly-hard RTS, one fundamental step is the selection of standard measurement units and the accurate determination of errors associated with different measurement units under test. In most cases, an error arises from the tolerance of the measurement unit [69] which represents the percentage of error that may deviate from the measured value. The values of uncertainty for standard component and test component are imperative to know for evaluating the calibration accuracy. We use the term measurement unit and measurement component alternatively throughout this work. Moreover, we assume that the measurement system model is known a priori to us.

- Consider we have different measurement units $M = \{M_1, M_2, \ldots, M_m\}$ in a RTS such that $m \in \mathbb{N}$.
- Each measurement unit under test $M_i \in M : (1 \leq i \leq m)$ is associated with its own accuracy Am_i where $Am = \{Am_1, Am_2, \ldots, Am_m\}$.
- The output of each measurement unit is Om_i where $Om = \{Om_1, Om_2, \ldots, Om_m\}$.
- Similarly, we assume the standard calibration unit S_i measures the output Os_i against the same input that is applied to the test unit maintaining an accuracy As_i where $S = \{S_1, S_2, \ldots, S_m\}$, $As = \{As_1, As_2, \ldots, As_m\}$ and $Os = \{Os_1, Os_2, \ldots, Os_m\}$ for which $m \in \mathbb{N}$.

To know the correct component accuracy, we also consider the tolerance ρ_j of each unit such that, $\rho = \rho_j : (0 \leq j \leq m)$. For example, the tolerance of a component $\rho_j = \pm 0.1\%$, means that the component output value will be in between 0.1% above or 0.1% below of the nominal value.

In our work, we compare all the measurement units under test (M) with the standard measurement equipment (S) to find the errors that are adjusted during the calibration. Let, the calibration error coefficient $Es_j \in Es$ of each standard component and $Em_j \in Em$ of test component where $Es = \{Es_1, Es_2, \ldots, Es_n\}$, $Em = \{Em_1, Em_2, \ldots, Em_n\}$; $n \in \mathbb{N}$ for which we observe different values in the measurement. We calculate the combined accuracy of standard calibration units (As), and the combined accuracy of measurement units under test (Am) to maintain the 4:1 calibration test accuracy ratio. Once the calibration process meets the TAR, we check the measured output with the tolerance range which assists in determining any unexpected behavior.

Definition 3.2. *Unexpected behavior: For a particular measurement unit* M_i*, we define an unexpected behavior as an event if* $Om_i \notin$ *[minimum value, maximum value] where the measured output* Om_i *associated with the tolerance. Such behavior indicates that there might be a problem in the measurement unit that may require calibration.*

b) Methodology for monitoring output:

Many embedded systems perform auto-calibration for their measurement units where an accuracy ratio of 4:1 is required during calibration. To calibrate the measurement unit of any real-time embedded system, we set a reference reading for comparison with the measured value. Most of the laboratory components are calibrated against the highest level of accurate measurement standards (e.g., NIST standards), but other standards exist for calibration are shown in Figure 3.12.

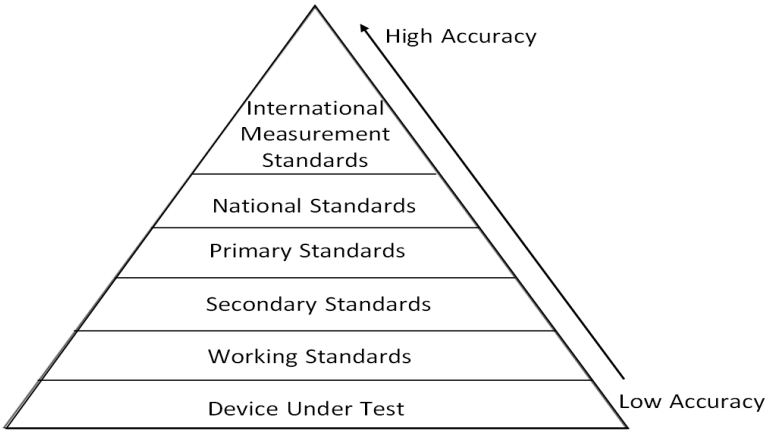


FIGURE 3.12: Types of measurement standards

Moreover, when a system performs calibration by setting the appropriate parameter search, it checks with different standard measurement units for the better calibration accuracy. At the same time, the system calculates the combined accuracy of measurement unit under test and standard measurement unit to maintain the 4:1 accuracy ratio. The accuracy of the component with multiple measurement units can be found by combining all the units accuracy to derive a single calibration component accuracy. Equations (3.16) and (3.17) show the calculation process of standard component accuracy (As) and device under test accuracy (Am). The accuracy values are calculated from the square root of the sum of the error (Es_i or Em_i) squares of each measurement units (M_i) [70].

$$As = \sqrt{(Es_1)^2 + (Es_2)^2 + \ldots + (Es_n)^2} \tag{3.16}$$

$$\mathrm{Am} = \sqrt{(\mathrm{Em}_1)^2 + (\mathrm{Em}_2)^2 + \ldots + (\mathrm{Em}_n)^2} \tag{3.17}$$

In the advancement of the technologies, many measurement component service suppliers provide the auto-calibration tools for which the system calibrates automatically when it is required. We consider the following two cases:

Case 1. ***Standard measurement unit is available:*** *In this case, we assume that the measurement component under test is compared against the attached standard measurement unit of known accuracy.*

To decide whether the measurement unit under test needs to calibrate or not, we check the measured value (Rm) with the accuracy range and suggest for calibration when the values exceed the range. Besides, we also examine the TAR to maintain a 4:1 ratio throughout the calibration. The methodologies for monitoring the calibration process are illustrated in Algorithm 3.

Algorithm 3 Auto-calibration monitor

```
1:  procedure MONITOR–CALIBRATION(Em, Es, Om, Os)
2:      for each (M_j, S_j ∈ M, S) : ∀(1 ≤ j ≤ n) do
3:          As_j ← find_standard_accuracy(Es)
4:          Am_j ← find_test_accuracy(Em)
5:          TAR_j ← test_accuracy_ratio(Am_j, As_j)
6:          exp_j ← expected output for each measurement unit
7:      end for
8:      for each (M_i, S_i ∈ M, S) : ∀(1 ≤ i ≤ n) do
9:          if (TAR_i ≠ user defined ratio) then
10:             if (Om_i ∉ [exp_i ± Am_i] & Os_i ∈ [exp_i ± As_i]) then
11:                 NotifyAdmin("warning") //log the data
12:             end if
13:         end if
14:     end for
15: end procedure
```

Example 3.1. *Consider that a sensor device has a measurement unit requiring* $\pm 6\%$ *measurement accuracy. It is recommended that the sensor device should be calibrated periodically to ensure the errors associated with the measurements remain within the acceptable range. Assume that the sensor receives a value 100 from the environment and the measurement unit is supposed to measure the same value.*

- *To satisfy Case 1, we consider a standard measurement unit with a known accuracy of* $\pm 2\%$*. According to Algorithm 3, we check the TAR (4:1) along with the measured*

value. A warning message will be displayed if the test device does not maintain the TAR and measures any value that exceeds the accuracy range (94 to 106).

Case 2. ***Standard measurement unit is unavailable:*** *In this case, we assume that the standard for the calibration process is unavailable, but we have other measurement components with known accuracy to read the output of the measurement unit.*

This particular case applies when a standard calibrated component is unavailable, and we still like to detect anomalies of measurements. To handle this case, we present a new approach to determine a safety bound to check whether a measurement unit under test measures values correctly or not after its first calibration. The defined bound is calculated from the tolerance of components for the measurement unit under test. The calculation steps are shown in Algorithm 4. This approach assumes that the measurement unit to be tested is previously calibrated correctly.

Algorithm 4 Measurement output monitoring.

```
1: Input: System input for each measurement unit
2: procedure OUTPUT–CHECKING(INPUT)
3:     for  each measurement unit do
4:         exp ← find_expected_output()
5:         Output[max, min] ← calculate_expected_output_range()
6:     end for
7:     for each measurement unit do
8:         measured ← get_measured_output()
9:         if (exp, measured) ∉ Output[max, min] then
10:            NotifyAdmin("warning") //log the data
11:        end if
12:    end for
13: end procedure
```

Example 3.2. *In the absence of calibration standard, we present a case study for monitoring measurement device. In this case study, we define a safety bound for the measurement unit to identify any unusual pattern. As an example, a resistive voltage divider is discussed below.*

c) An illustrative monitoring example for calibration:

In RTSs, usually, we see the unavailability of a standard measurement unit which is required for the purpose of equipment calibration. Therefore, in this work, we emphasize Case 2, leaving the other cases to be demonstrated for future work. As an example, we discuss a resistive voltage divider. We focus on the 25-bit reference voltage [71] divider

that uses switching techniques [72][73] for self-calibration in multiple stages. The only way to test the voltage divider is to plug the whole unit where the calibration result cannot be perceived until the completion of multiple stages. We implement monitors at each stage to notify us if any unexpected behavior occurs at any stages. The rationale behind choosing the voltage divider as a case study is to understand the correctness of measurements easily and estimate the range of standard values.

Resistance Measurement: Consider we have a voltage divider that distributes an input voltage V into multiple stages $L = \{L_1, L_2, \ldots, L_m\}$ such that $m \in \mathbb{N}$. Each stage $L_i \in L : (1 \leq i \leq m)$ consists of n number of resistors $R = \{R_1, R_2, \ldots, R_n\}$ where $n \in \mathbb{N}$. Each resistor R_j, has a tolerance ρ_j such that, $\rho = \rho_j : (0 \leq j \leq n)$. For each resistor R_j, we calculate the approximate actual resistance Ro_j and the measured resistance Rm_j where $Ro = \{Ro_1, Ro_2, \ldots, Ro_n\}$ and $Rm = \{Rm_1, Rm_2, \ldots, Rm_n\}$ respectively. To find the approximate actual resistance, we consider an error coefficient $E_j \in E$ where $E = \{E_1, E_2, \ldots, E_n\}$ for which we observe different values in the measurement. We estimate the maximum (E_{max_j}) and minimum (E_{max_j}) error coefficient associated with the resistor R_j, and choose an average value (E_{avg_j}) of these two which are used to calculate the approximate actual resistance.

Voltage Measurement: Once we determine the approximate actual resistance, we can also calculate the expected voltage using Ohm's law for each stage of the voltage divider where the input voltage source is V, and the current is I. The expected output voltage for ith stage,

While measuring the voltages through each resistor, we consider the measurement component M_j which has an accuracy Av. In auto-calibration process for m stage resistive voltage divider, we measure output voltage $Vm = \{Vm_1, Vm_2, \ldots, Vm_n\}$ and assume that the actual voltage drop at each stage is Vo_j against resistor Ro_j. Here actual voltage drop, $Vo = \{Vo_1, Vo_2, \ldots, Vo_n\}$. Since the actual voltage drop does not match the measured voltage due to multimeter measurement accuracy, we determine a voltage range across each resistor to monitor any fault occurred during calibration. For each actual voltage drop Vo_j, we calculate the maximum and minimum possible value V_{max_j}, V_{min_j} respectively.

Proposed workflow for a resistive voltage divider: For each resistor R_j with tolerance ρ_j, we determine a possible maximum and minimum resistance using following calculations.

Possible maximum resistance, $R_{max_j} = R_j + (R_j \times \rho_j)$

Possible minimum resistance, $R_{min_j} = R_j - (R_j \times \rho_j)$

We measure the resistance (Rm_j) for each resistor R_j using the measurement component. Apart from calculating the original value of each resistor, we consider the associated error coefficient E_j and the measured resistance (Rm_j) which can be calculated using Equation 3.18.

$$Rm_j = Ro_j \times E_{avg_j} \Rightarrow Ro_j = \frac{Rm_j}{E_{avg_j}} \tag{3.18}$$

According to tolerance ρ_j, associated with a resistor R_j, we know that the actual value of any resistor should be in between R_{max_j} and R_{min_j} which indicates that the error coefficient of any resistor should have the following bound.

$$E_j \in [E_{min_j}, E_{max_j}] = E_j \in [\frac{Rm_j}{R_{max_j}}, \frac{Rm_j}{R_{min_j}}] \tag{3.19}$$

To calculate E_{avg_j} as accurately as possible, we take an average of E_{min_j} and E_{max_j}. Alternatively, Ohm's Law derives the approximate expected output voltage at each stage of the voltage divider. We also define a voltage drop range using Equation 3.20 which considers measurement component accuracy.

$$(1 - Av)Vo_j \leq Vm_j \leq (1 + Av)Vo_j \tag{3.20}$$

where $\forall\, j \in N;\ 1 \leq j \leq n$. From Equation 3.20, we calculate a minimum and maximum bound for actual voltage drop in any stage. In this work, we show that the specified voltage range for any auto calibration process is always reliable for any stage and the experimental data holds this assumption correctly.

Once the output voltage of any stage goes beyond the defined range, we assume that there is a possibility of an unusual event at that stage. On the contrary, if the output voltage always remains within the specified scale, we conclude that the resistors and the measurement units are working correctly. The monitoring process of all stages is illustrated in Algorithm 5. In this software-based calibration approach, we show a case study of defining a standard for a system component which requires to maintain a particular accuracy or output bound. The calculated output bound may vary from component to component but the calibration approach will be similar to all cases.

Algorithm 5 Voltage divider output monitoring.

procedure VOLTAGE–MEASUREMENT(R_o, V, Av, Vm)
  $I = \frac{V}{\sum_{j=1}^{n} Ro_j}$
  for each $(V_j \in V) : \forall(1 \leq j \leq n)$ **do**
    $Vo_j \leftarrow$ find_expected_voltage(Ro_j, I)
    $[V_{max_j}, V_{min_j}] \leftarrow (1 - Av)Vo_j \leq Vm_j \leq (1 + Av)Vo_j$
  end for
  for each $(L_i \in L) : \forall(1 \leq i \leq n)$ **do**
    if $(Vo_i, Vm_i) \notin [V_{max_i}, V_{min_i}]$ **then**
      NotifyAdmin("warning") //log the data
    end if
  end for
end procedure

3.4.2.5 Discussion

The main purpose of this calibration framework is to ensure the accuracy of the task output through the monitoring approach. In this work, we use the RM scheduling to understand the timing effects on the low QoS requirement tasks from high QoS tasks. The RM scheduling condition provides the timing interference that helps to decide whether a task is schedulable or it is manageable to schedule by changing its timing properties to be executed based considering available resources.

Chapter 4

Experimental results and analysis

4.1 Goals of the experiments

In our experimental work, we conduct several experiments to illustrate the applicability of the proposed framework regarding the requirements preservation in soft and weakly-hard RTSs. The goals of our experiment are to analyze the proposed design automation and adaptation approaches for meeting the requirements. Therefore, we perform an experiment using the OpenMP parallel programming framework to meet the QoS requirements for soft RTS applications. At the same time, we apply a calibration framework that includes the LITMUS^{RT} tracing mechanism for monitoring the task output to meet accuracy-based QoS requirements in weakly-hard RTSs.

4.2 Analysis of design automation and requirements preservation in soft RTSs

The main objectives of this experiment are: (1) understand the necessity of design automation in for implementing OpenMP parallel programs, (2) analyze the deterministic behavior of a task execution using different task scheduling approaches, (3) examine the OpenMP scheduling techniques in terms of varying requirements of tasks, (4) analyze the execution time overhead for existing loop scheduling approaches in OpenMP and (5) visualize the performance of our proposed parallel loop-based scheduling approach in terms of missing task execution time-bound. We demonstrate the applicability of our experimental goals using several scenarios described as follows:

4.2.1 Importance of design automation in writing parallel programs

We use the COCOMO [74] cost model to understand the necessary effort for implementing an OpenMP parallel program in terms of lines of code. The reason behind the use of the COCOMO model is because of its procedural cost estimation model that is used as a process of reliably predicting the various parameters associated with embedded systems. On the other hand, existing cost estimation models (e.g., agile and waterfall) do not specify any constraints for embedded system applications. However, in this work, we calculate and compare the effort of our proposed AADL based automated design process.

To write a parallel program, a developer needs to write a program code that may differ with sizes of the software. With the expansion of the lines of code in writing a parallel program, it requires more effort and time which increase the cost of developing embedded applications. Automation in the design and implementation process for writing a parallel program can reduce the overall cost. In this experiment, we use the Constructive Cost Model (COCOMO) to estimate the costs (e.g., effort and duration) to write a parallel program including a different number of tasks.

TABLE 4.1: Cost estimation using COCOMO for writing OpenMP parallel programs

LOC	**A**	**B**	**C**	**D**	**Effort (in hours)**
80	3.6	1.2	2.5	0.32	14.59
100	3.6	1.2	2.5	0.32	15.26
145	3.6	1.2	2.5	0.32	18.33
240	3.6	1.2	2.5	0.32	22.24
278	3.6	1.2	2.5	0.32	23.53
310	3.6	1.2	2.5	0.32	24.54
400	3.6	1.2	2.5	0.32	27.06
450	3.6	1.2	2.5	0.32	28.32

The COCOMO model is a good measure for estimating the effort (e.g., the amount of effort and time) required to develop a program. The amount of effort (effort = $A * \text{LOC}^B$) is measured in person per month to complete a task by writing the number of lines of code (LOC). This amount of effort can be converted to time using the specified constant variables that are determined by the COCOMO model. We also calculate the required effort in time (time = $C * (\text{Effort})^D$) for writing a program where A, B, C, and D are constant variables [75]. For a different LOC (differ in the number of tasks), we show the

results of effort in Table 4.1 and Table 4.2 for writing the OpenMP parallel programs and AADL based programs.

TABLE 4.2: Cost estimation using COCOMO for writing AADL Code

LOC	**A**	**B**	**C**	**D**	**Effort (in hours)**
50	3.6	1.2	2.5	0.32	12.18
90	3.6	1.2	2.5	0.32	15.27
130	3.6	1.2	2.5	0.32	17.58
138	3.6	1.2	2.5	0.32	17.99
145	3.6	1.2	2.5	0.32	18.33
160	3.6	1.2	2.5	0.32	19.04
200	3.6	1.2	2.5	0.32	20.74
210	3.6	1.2	2.5	0.32	21.14

We measure the effort in time to write the same program for a different number of tasks using AADL specification. After that, we compare the amount of effort for writing both in OpenMP and AADL which is shown in Figure 4.1. We observe that the amount of effort for writing a lengthy parallel program in AADL specification requires less time (hours) than the OpenMP parallel code. Thus, it indicates the importance of design automation conforming AADL approach for implementing an extensive sizeable parallel program.

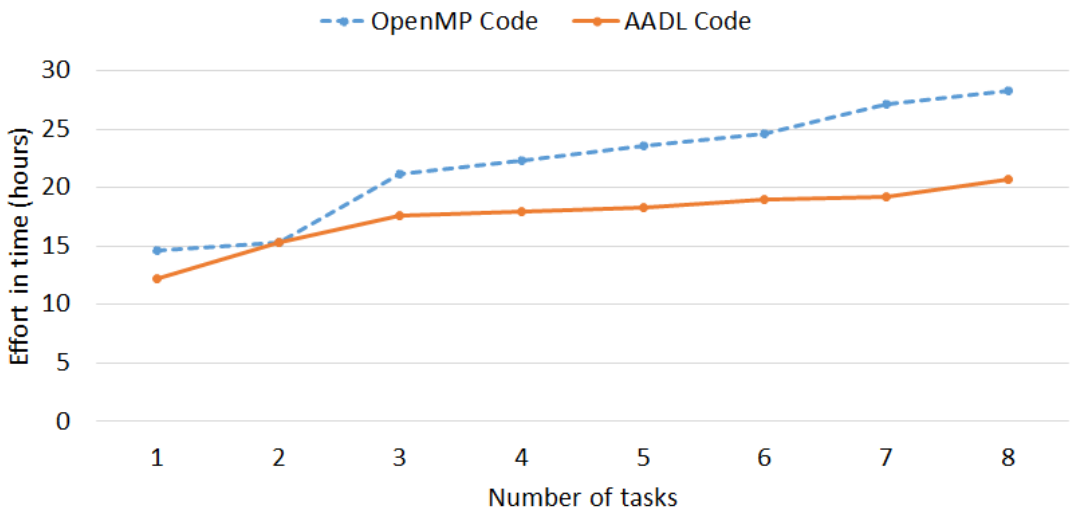


FIGURE 4.1: Estimated efforts comparison between a OpenMP parallel program code and an AADL based design Code.

4.2.2 Analysis for deterministic task execution using OpenMP

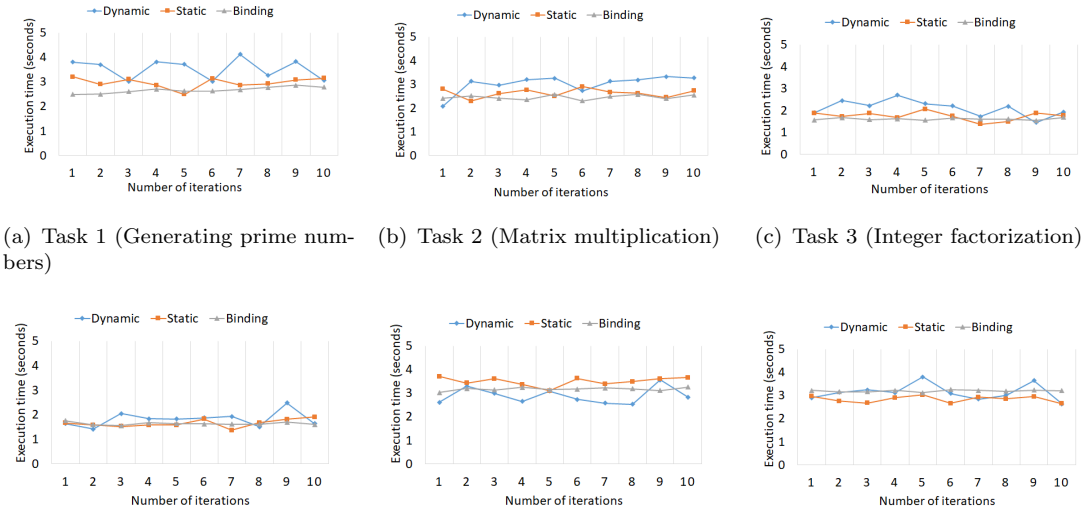
In this test scenario, we capture the execution times of different tasks applying the OpenMP task scheduling approaches and explore the deterministic behavior in producing the same result.

To understand the execution behavior in between the OpenMP static and the thread to processor binding approaches, we measure the standard deviation of execution times. The standard deviation [76] ($\sigma = \sqrt{\frac{\sum_{i=1}^{N'}(e_i-\mu)^2}{N'}}$) of execution times partially represents the determinism of a scheduling approach where where e_i is an individual task execution time, μ is the mean of the sample execution times and N' is the total number of executions of each task. In addition, we calculate the values in the confidence interval range, $CI(\%) = \bar{e} \pm z^* \frac{\mu}{\sqrt{N'}}$) of 95% where z^* is the upper critical value for the standard normal distribution and $\bar{e}$ is equal to the mean (μ) of the sample execution times [76]. This confidence interval is traditionally used to calculate an estimated range of sample values which is likely to include an unknown population. However, several key factors such as synchronization overhead of threads and limited computational capacity of the processor are observed because these factors influence the overall task execution process. Analyzing these facts for each $\tau_{\mathrm{i}} \in \tau_{\mathrm{high}}$, we use the static scheduling approach assigning a thread to the targeted processor with the help of CPU_AFFINITY($\mathrm{P_i}$) method. We determine that the thread to processor binding provides significant advantages at run-time synchronization and increases the predictability of the system.

To analyze the task execution times, we create *six* independent tasks defined as τ_1 (Generating prime numbers), τ_2 (Matrix multiplication), τ_3 (Integer factorization), τ_4 (Integer Sort), τ_5 (Fibonacci sequence generation), and τ_6 (Pi digit calculation). We run all these *six* tasks for more than *ten* times in parallel against the static, the dynamic and the thread to processor binding approaches.

In this experiment, we use a multiprocessor system which has four logical processors, 64-bit Ubuntu 16.04 operating system, and 5.4.0 GCC compiler which supports OpenMP 5.0 version to run C/C++ programs. Here, the system contains the following configurations: processor Intel i5 4210M with Clock Speed 2.9 GHz, RAM 4.00 GB, Cache Size 512 KB and Linux kernel version 4.4.0-111.134.

While running all the tasks multiple times, we observe that the execution times vary from each other in different scheduling approaches. In the case of static and dynamic



(a) Task 1 (Generating prime numbers)

(b) Task 2 (Matrix multiplication)

(c) Task 3 (Integer factorization)

(d) Task 4 (Integer Sort)

(e) Task 5 (Fibonacci sequence generation)

(f) Task 6 (Pi digit calculation)

FIGURE 4.2: OpenMP task execution time differences among dynamic, static, and the thread to processor binding approaches

scheduling approaches, the variance on the execution times for each task is too high which makes these approaches almost non-deterministic. Figure 4.2 shows the execution time variation of each task for different scheduling approaches. The system exhibits different execution time in each run with high inconsistency for both dynamic and static scheduling approaches. In comparison to these methods, we observe that the thread to processor binding approach shows better deterministic execution time and less inconsistency for each task. Figure 4.3 presents the standard deviation of execution times for each task where the thread to processor binding approach shows on average 12% improvement that indicates the more deterministic execution than the current static approach. We observe that the variance of the execution time for each task is smaller in the proposed binding approach compared to the static approach, with the 95% confidence interval range. The smaller variance of execution time implies the improvement of deterministic performance for high QoS tasks.

4.2.3 Overhead analysis of the proposed binding approach

In this test scenario, we measure the overhead for executing the proposed thread to processor binding approach over the dynamic approach. We examine the task overhead by creating a different number of threads at runtime.

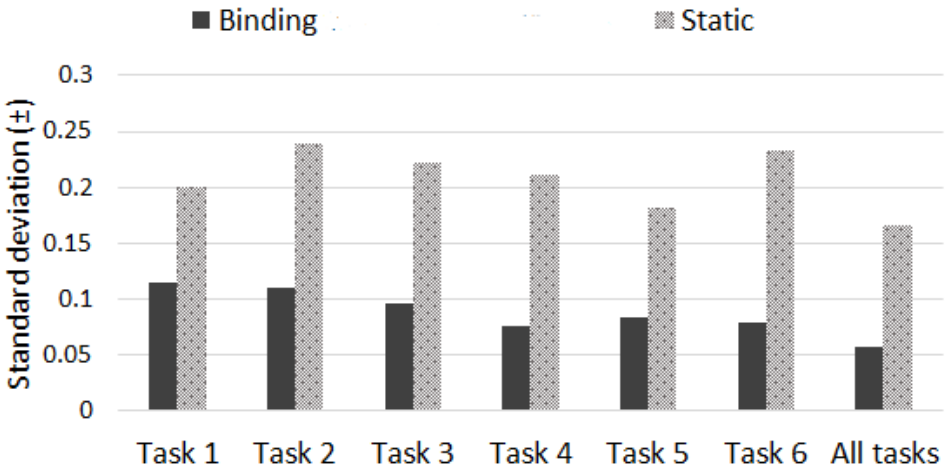


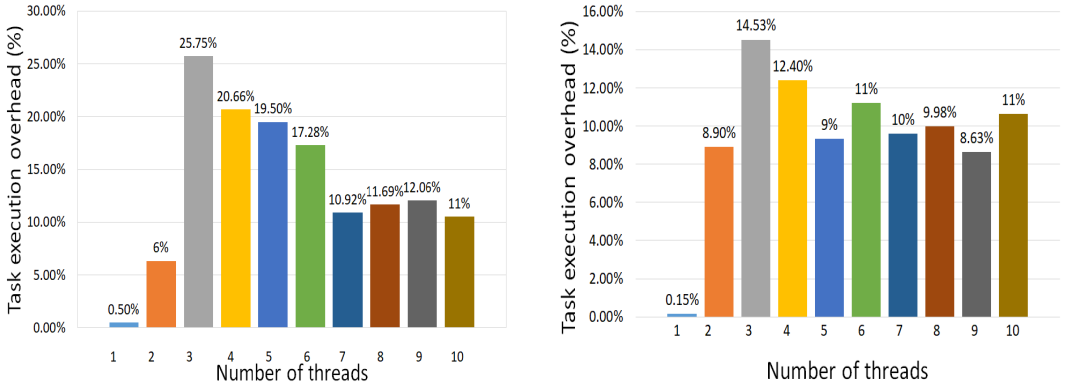
FIGURE 4.3: Variation of execution times between the static approach and the thread to processor binding approach

To analyze the performance of OpenMP task parallelism techniques in terms of task execution time, we create *ten* individual tasks and run them more than 50 times to visualize the execution time variation. In this experiment, we release all the tasks at the same time and run on the same quad-core processor system which is defined earlier. The experimental results show the comparison of tasks execution time differences for both the thread to processor binding and the dynamic scheduling approach.

We calculate the overhead using the execution time variances between dynamic and binding approach. The overhead Θ is calculated using $(\frac{E_T^{binding} - E_T^{dynamic}}{E_T^{dynamic}}) * 100$, which is based on execution time for running tasks using the thread to processor binding approach over the dynamic approach. We calculate the total execution time $E_T^{binding}$ and $E_T^{dynamic}$ by running all tasks using the binding and dynamic approach respectively.

To compare the overhead between the proposed binding approach (used for high QoS tasks) and dynamic scheduling approach (used for low QoS tasks), we create a number of threads starting from *one* to *ten* and execute the tasks using both threads to processor binding and dynamic scheduling approaches. We run the experiment 20 times against the assigned number of threads and measure the total execution time at each run. To simplify the overhead calculation, we consider the maximum or worst-case execution times for $E_T^{binding}$ and $E_T^{dynamic}$ against the defined number of threads.

Therefore, Figure 4.4 (a) shows the percentage of overhead for running all the tasks using the thread to processor binding approach where we create a different number of threads at the runtime. In this figure, we observe that the overhead has an increasing pattern



(a) Overhead on using the binding approach in laptop (processor Intel i5)

(b) Overhead on using the binding approach in Raspberry Pi 3 (processor ARM Cortex-A53)

FIGURE 4.4: Overhead on using the proposed binding approach compared to the dynamic approach with varying number of threads

compared to the dynamic approach until the number of threads equal to 3. However, the overhead starts to decrease when the specified number of threads become equal to the number of processors. In our experiment, we observe that the total execution time in binding approach remains almost unchanged even we increase the number of threads, after it becomes equal to the number of processors. We also find an increased task execution time using dynamic scheduling approach when we create an unnecessary number of threads that exceed the number of processors. Figure 4.4(a) shows that the overhead Δ of the proposed binding approach is low compared to the dynamic approach, when the number of threads is more than the number of processors.

Task execution time overhead in Raspberry Pi: To evaluate the tasks scheduling approaches in soft real-time embedded systems, we execute the same *ten* tasks in Raspberry Pi 3 with the following specifications. It has Broadcom BCM2837 SOC, 4×ARM Cortex-A53 1.2GHz CPU, 1GB LPDDR2 (900 MHz) RAM, Cache Size 256 KB and Debian Linux Kernel on Ubuntu 16.04.

Figure 4.4(b) shows the overhead Δ for the proposed binding approach over the dynamic task scheduling approach. In this case, similar to the previous experiment, we observe that the overhead in the binding approach starts to decrease after an initial increase when the number of active threads becomes four. Moreover, this also indicates that the dynamic approach starts to perform poorly when the number of threads is becoming more than the number of processors.

4.2.4 Analysis on requirements preservation of the proposed thread to processor binding approach

Again, we assume a set of *ten* tasks, where each task has an individual requirement and execution time-bound. The proposed framework schedules all the high QoS and low QoS tasks using the thread to processor binding and the dynamic approaches respectively, so that the high QoS tasks can meet their required execution time-bound.

We define τ_1, τ_2 and τ_3 as high QoS tasks and the remaining tasks are as low QoS tasks. According to our proposed approach, all three high QoS tasks are scheduled first using the thread to processor binding approach, and the remaining tasks are scheduled dynamically based on the availability of the processors. Besides, to show a comparison with the proposed approach, we run all the tasks dynamically for *ten* times without considering any requirements of the tasks.

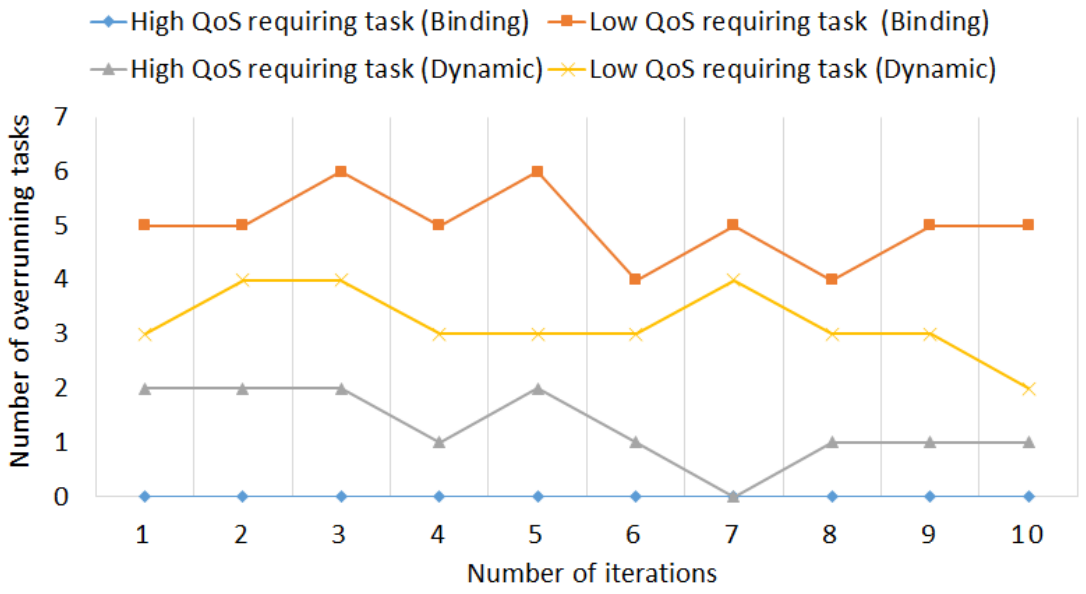
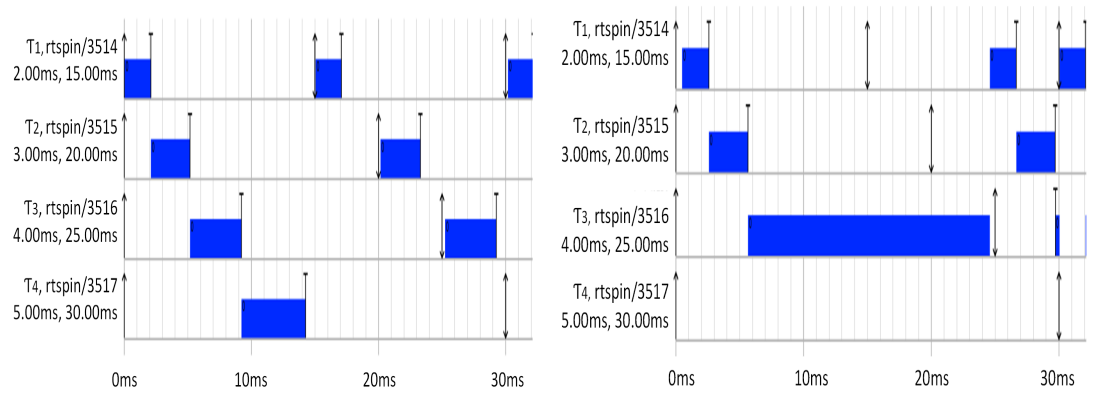


FIGURE 4.5: Number of overrunning tasks in binding (thread to processor) and dynamic scheduling approach

We observe that the number of overrunning tasks in both the binding approach and the dynamic approach is almost similar. However, the proposed approach outperforms the dynamic approach in terms of the number of overrunning high QoS tasks. In the proposed thread to processor binding approach, the number of overrunning high QoS tasks is zero as presented in Figure 4.5. We also observe that the high QoS tasks start to overrun when they are scheduled dynamically.

4.3 Monitoring accuracy for requirements preservation in weakly-hard RTSs

The experiment involves implementing a prototype that runs on LITMUS^{RT} 4.1.3+ [13]. The LITMUS^{RT} is a real-time extension of the 3.13.6 linux kernel version that focuses on real-time task scheduling. To control the CPU-level frequency, we install LITMUS^{RT} inside the Linux operating system (OS) on an Intel Core i5 processor that supports frequency scaling up to 1.8GHz. We create a task model consisting of four tasks $\tau_1[2, 15, 15, 1], \tau_2[3, 20, 20, 2], \tau_3[4, 15, 25, 3]$, and $\tau_4[5, 30, 30, 4]$ where each task is defined sequentially with an execution time (ms), a deadline (ms), a period (ms) and a priority value.



(a) A scenario when tasks remain schedulable

(b) Tasks (τ_3, τ_4) miss the deadline for inaccurate output voltage

FIGURE 4.6: Timing diagrams to determine tasks schedulability for different supply voltages

To evaluate our proposal, we consider a resistive voltage divider as a hardware component that turns a supply voltage to an expected output voltage in DVFS. In our experiment, we assume that the acceptable accuracy for converting output voltage is $20 \pm 2\%$ which means the task set remains schedulable as long as the output voltage is inside the accuracy range. The LITMUS^{RT} library 'rtspin' is used to create four real-time dummy processes choosing partitioned multiprocessor Rate Monotonic (RM) algorithm where the estimated CPU utilization (0.61) gives the opportunity to modify the timing properties of tasks. We scale down the voltage supply and frequency level into different discrete levels. Figure 4.6 (a) shows that the tasks always remain schedulable for a particular voltage supply (20V). However, the CPU utilization starts to increase when the supply voltage is scaled down. Since the required execution time changes with

the supply voltage, the task set exceeds the maximum CPU utilization ($U_{max} \simeq 1$). Figure 4.6 (b) shows that the task set fails to pass the schedulability test as τ_3 takes extra time to complete its process.

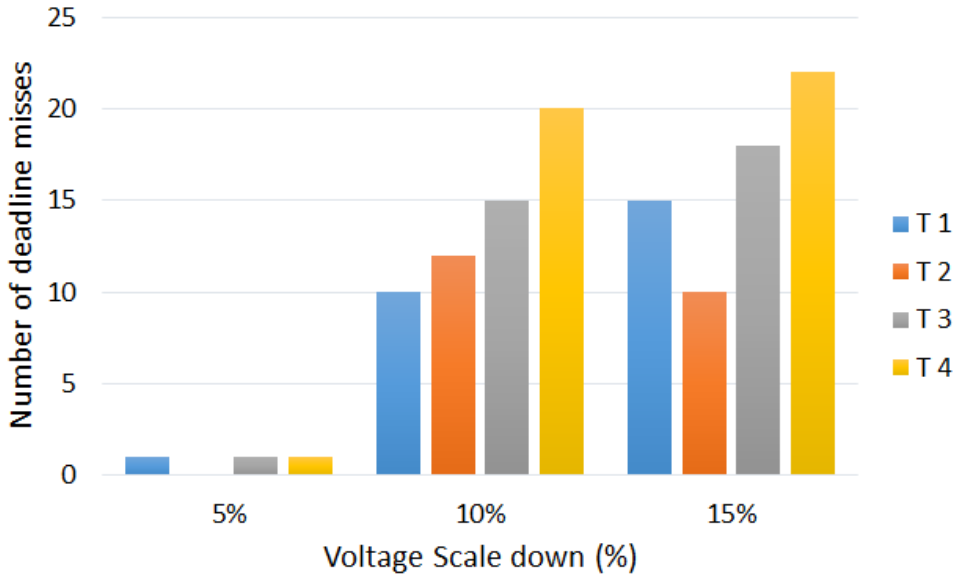


FIGURE 4.7: Number of deadline misses for different voltage scale levels

```
Task,   Job,     Period,   Response, DL Miss?,   Lateness,  Tardiness, Forced?,       ACET
task NAME=rtspin PID=4919 COST=2000000 PERIOD=15000000 CPU=0
4919,     2,   15000000,       8133,        0,  -14991867,          0,       0,       7873
4919,     3,   15000000,    2413751,        0,  -12586249,          0,       0,    2099518
4919,     4,   15000000,    2210347,        0,  -12789653,          0,       0,    2042979
4919,     5,   15000000,    2108356,        0,  -12891644,          0,       0,    2048350
4919,     6,   15000000,    2486352,        0,  -12513648,          0,       0,    2049221
4919,     7,   15000000,    2445707,        0,  -12554293,          0,       0,    2075156
4919,     8,   15000000,   17453805,        1,    2453805,    2453805,       0,   17196668
4919,     9,   15000000,    4499898,        0,  -10500102,          0,       0,    2042643
```

FIGURE 4.8: A LITMUS trace showing response time, deadline miss, tardiness (delay) and actual worst-execution time (ACET) of different tasks

4.3.1 Analysis of task schedulability test

To understand the effect of correct output voltage production in weakly-hard real-time task scheduling, we run the task set for $5ms$ after scaling down the supply voltage in different percentage. Calculating the maximum available CPU utilization, we increase the task execution time and the deadline of a task using Equation 3.11 and 3.12. With the

new estimation of e_i and d_i, we observe that our defined tasks never miss the deadlines while the scaling level of supply voltage remains in between 0 to 4%. Figure 4.7 shows the number of missed deadlines for each task against different voltage scaling levels that indirectly change the CPU clock frequency. Hence, any system requiring maximum 4% voltage scaling for a specific supply voltage may experience a system failure if the associated resistive voltage divider is unable to produce an output within this range. In Figure 4.7, the ratio of deadline miss for τ_4 is comparatively higher than others as the high priority tasks preempt the low priority tasks in rate monotonic scheduling policy. However, the incorrect output simplifies the importance of a software-based monitoring module that checks the accuracy of the system components in task scheduling.

4.3.2 Task output analysis from recorded trace in LITMUS$^{\mathbf{RT}}$

The LITMUS$^{\mathrm{RT}}$ tracing mechanism which includes feature-trace and sched-trace is applied for monitoring and acquiring task schedules. We record all the response time, deadline miss, overhead, synchronization delay, CPU frequency and actual execution time for each task during a specific supply voltage. Figure 4.8 shows a sample of recorded traces for task τ_1 (identified by process id 4919) where a flag is raised when it misses the deadline. The initial traces are recorded as the standard output assuming the system components are performing correctly with the expected accuracy. Afterward, the new inputs are matched with the previous trace pattern to identify the mismatch against the expected output. Moreover, the QoS of a task depends on the task output that can be measured from the recorded traces. As an example, we analyze the recorded system throughput during the task execution. Figure 4.9 illustrates a trade-off between the voltage scale down and the system throughput. It shows the different throughput collected for different voltage down scaling levels. This throughput indicates the QoS that can be compared with the previously defined QoS range to visualize the overall system performance.

4.3.3 Error correction for calibration

To handle the output inaccuracy of a system component, we run four different periodic tasks $\tau_1[2, 18, 18, 1]$, $\tau_2[4, 20, 20, 2]$, $\tau_3[5, 22, 22, 3]$, and $\tau_4[5, 25, 25, 4]$ in the same environment that is mentioned earlier. We log all the output of the first cycle (hyper period 25ms) for executing the task set. Thus, we consider these trace data as a calibration

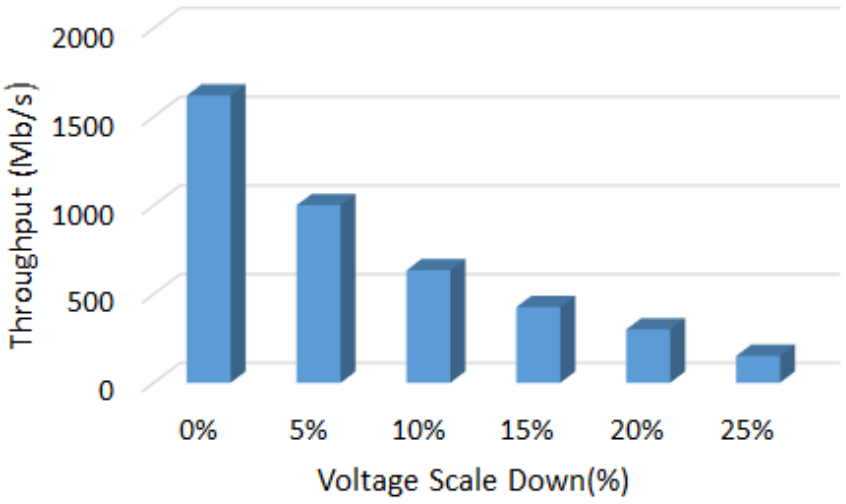


FIGURE 4.9: Throughput analysis for different voltage scale levels

standard because we see it continues following the same data pattern without violating any deadline. We let the experiment run for a long time and start tracing its output. Figure 4.10 shows the example of a calibration standard that is used for matching data pattern in each task execution cycle. After more than 140 cycles, we observe that the task τ_4 start missing its deadline. At this point, we calculate the CPU utilization $(n * (2^{1/n} - 1) \approx 0.74)$ which is still less than the RM maximum bound 0.75. That indicates that the task set is not schedulable for the inaccuracy of a system component. Now, this situation can only occur due to different issues of a software component or a

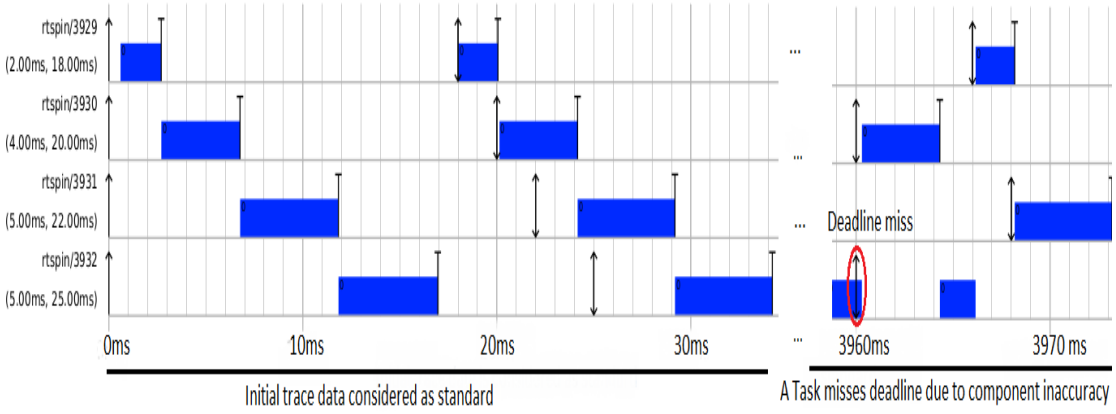


FIGURE 4.10: Deadline miss of a system component on LITMUS^{RT} even though tasks meet the utilization bound (a 74% use of the CPU)

hardware component. However, initially, we assume that the inaccuracy of the output is due to a software component.

Software rejuvenation [77] is one of the recognized techniques for mitigating performance deterioration effects of a software component which may occur for software aging. Thus, we apply the software rejuvenation technique to correct the error and run again to review the output. This calibration process shows it starts with an expected trace data pattern without any problem. However, if the software component is unable to resolve the error, our framework suggests to calibrate the hardware component with a high accuracy standard. Finally, the experimental result states the demand for a calibration framework that notifies the system when a system component requires calibration.

4.3.4 Analysis of software-based calibration in a resistive voltage divider

In the experiment, we create a prototype of a 3-stage resistive voltage divider and verify the calculated results from different aspects. We use three resistors R_1, R_2, and R_3 connected in series

`Resistor:` The vendor name of the used resistors is Vishay Dale (VA) which claims a tolerance of $\pm(0.1)$ with temperature coefficient 25ppm / °C(temp).

`Multimeter:` We use TP 4000ZC Multimeter to measure the resistance and voltage where the measurement accuracy is $\pm 1.2\%$ for 0 to 4MΩ and $\pm 0.5\%$ for 0 to 400V respectively.

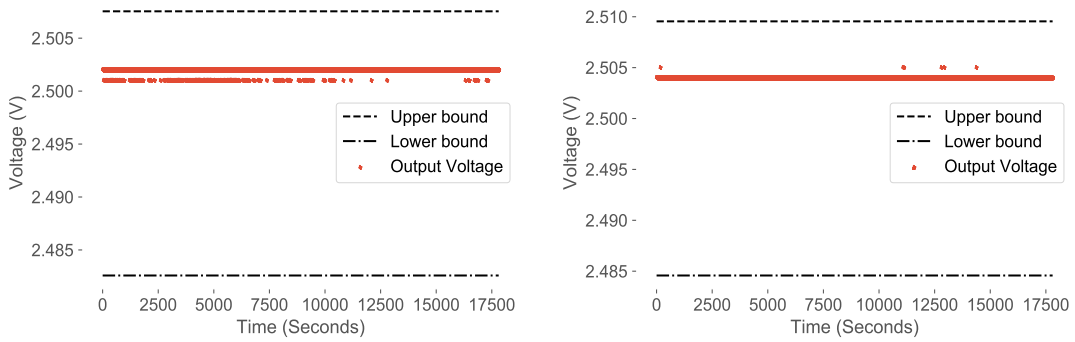
To demonstrate a brief overview of calculation, first, we examine the resistor, $R_1 = 40.2\text{k}\Omega \pm 0.1\%$ which has,

$$\text{Possible maximum resistance, } R_{\text{max}_1} = 40.24\ \text{k}\Omega$$
$$\text{Possible minimum resistance, } R_{\text{min}_1} = 40.16\ \text{k}\Omega$$

At the time of resistance measurement, the measured value may deviate from the actual resistance due to multimeter accuracy consideration. To find out the approximate readings for the first stage, we use different parameters like Error coefficient E_1, Actual Resistance Ro_1 and Measured Resistance $Rm_1 = 40.31\text{k}\Omega$.

Using Equation 3.18, Ro_1 can be computed as follows,

$$Ro_1 \times E_1 = Rm_1$$
$$\Rightarrow Ro_1 = \frac{Rm_1}{E_1}$$



(a) Voltage V_1 across resistor R_1 (b) Voltage V_2 across resistor R_2

FIGURE 4.11: Analysis of the monitored data for resistive voltage divider

The possible actual maximum value for this resistance is 40.24 kΩ and minimum is 40.16 kΩ which means E_1 should be greater than 1. Error coefficient calculation for R_1 can be given as,

$$Emin_1 = \frac{Rm_1}{Rmax_1} \qquad Emax_1 = \frac{Rm_1}{Rmin_1}$$

$$E_1 \in [\min, \max] = E_1 \in [1.00173956, 1.00374]$$

To find a single error coefficient from the above range, we take an average of the minimum and maximum value instead of choosing any random value. In this case, the calculated average value is $E_{avg_1} = 1.0027898$ and the estimated original resistance for R_1,

$$Ro_1 = \frac{40.31}{E_{avg_1}} k\Omega = 40.1978 k\Omega$$

Similarly, we calculate the original resistance for other two resistors, $Ro_2 = 40.20015 k\Omega$ and $Ro_3 = 40.19780 k\Omega$

Observation:

We calculate the actual current (I) to obtain the expected output voltage through each resistor.

$$I = \frac{V}{Ro_1 + Ro_2 + Ro_3} = \frac{7.52V}{120.59575 k\Omega}$$
$$= 0.062357 mA$$

Expected voltage drop across each resistor,

$$\mathrm{Vo_1} = \mathrm{I} \times \mathrm{Ro_1}$$

Total voltage: $\mathrm{Vo_1} + \mathrm{Vo_2} + \mathrm{Vo_3} = 7.51998917\mathrm{V} \approx 7.52\mathrm{V}$. The total expected approximate voltage drop over each stage produces the same input voltage combinedly.

Voltage range calculation:

From Equation 3.20, we define the voltage drop bound across first resistor where measured voltage is $\mathrm{Vm_1}$ and accuracy $\mathrm{Av_1} = \pm 0.5\%$ which can be given as,

$$0.995\mathrm{Vo_1} \leq \mathrm{Vm_1} \leq 1.005\mathrm{Vo_1}$$

$$\text{So, } \mathrm{Vo_1} \leq \left(\frac{\mathrm{Vm_1}}{0.995} = \frac{2.495}{0.995}\,V\right) \qquad \text{Or, } \mathrm{Vo_1} \geq \left(\frac{\mathrm{Vm_1}}{1.005} = \frac{2.495}{1.005}\,V\right)$$

Therefore, we summarize the voltage drop bound for each stage of resistive voltage divider following the above calculation.

$$\mathrm{V_1}[\min, \max] = [2.482587\mathrm{V}, 2.507537\mathrm{V}] \tag{4.1}$$

$$\mathrm{V_2}[\min, \max] = [2.484577\mathrm{V}, 2.509547\mathrm{V}] \tag{4.2}$$

$$\mathrm{V_3}[\min, \max] = [2.482587\mathrm{V}, 2.507537\mathrm{V}] \tag{4.3}$$

We show that the output voltage at any stage of the resistive voltage divider maintains the proposition as defined by Equations (4.1) - (4.3).

Results analysis:

To evaluate the calculated range as presented in Equations (4.1) to (4.3), the output voltage of each stage is logged for five hours with a one-second interval. Figure 4.11 shows the changes of the output voltages across resistor $\mathrm{R_1}$ and $\mathrm{R_2}$ in a room temperature of approximately 23.4°C where we carefully examine and control all the parameters that can have an impact on the change of properties of a resistance. According to the first stage voltage range statement, the output voltage should remain in between 2.482587V and 2.507537V where Figure 4.11(a) depicts the evidence of it. Similarly, we observe the output voltage from Figure 4.11(b) and it also follows the defined output bound for the second stage. As we see too many data points with the one-second interval in both Figure 4.11(a) and Figure 4.11(b), the solid line indicates where most of the points are

located. Since the resistance of R_1 and R_3 are the same, we get almost equal output voltage for each case. We also analyze the measurement accuracy after examining the overall measured output voltage. Figure 4.12 shows the divergence of measurement from the expected output voltage where expected_voltage $\pm$ .2% represents the accuracy of the voltage measurement for each stage.

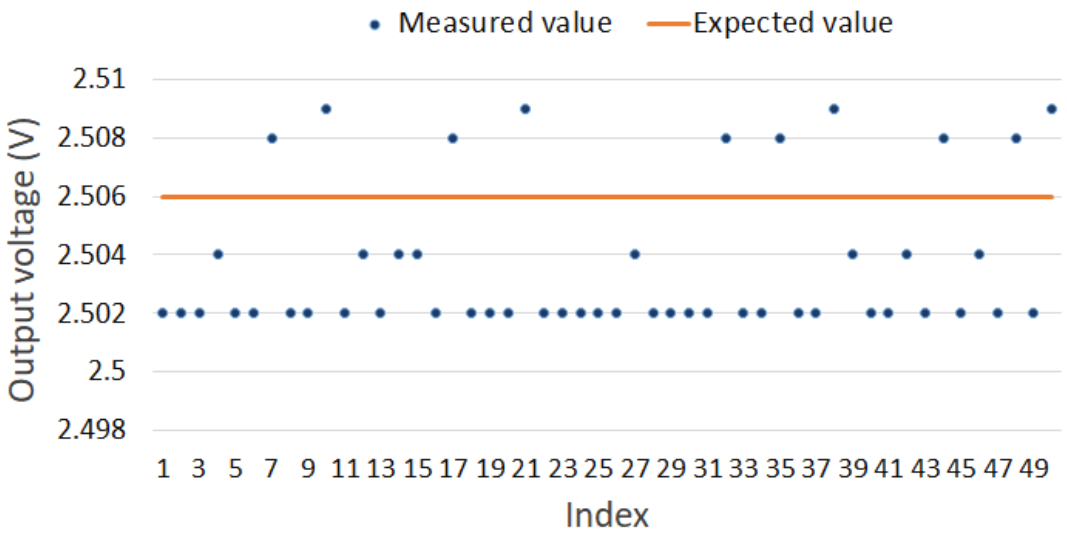


FIGURE 4.12: Accuracy analysis of measured voltage

The experimental results demonstrate that the values at each stage are within the defined voltage bound which ensures that all the resistors are in stable condition. At the same time, we show that any event outside this specified range indicates the possibility of fault occurred at that stage. Once we identify the stage that has the inconsistency, we notify the user to check that specific stage of the voltage divider.

Discussion:

The correctness of real-time embedded systems depends on the delivery of the appropriate value at the right time. Currently, in most cases, we prioritize the timing correctness but ignore considering the effect of correct measurements of the output. To produce the correct result consistently, calibration of measurement units is necessary to obtain the expected accuracy. This work presents a software-based approach to monitor the anomalies of measured values and uses the calibration considering the test accuracy ratio (TAR). As a proof-of-concept, we consider a voltage divider as a case study where the calibration standard is estimated by our new approach to monitor the overall system performance. The estimated standard voltage bound helps us to identify the faulty resistor or measurement units at runtime by analyzing the measured data patterns.

Chapter 5

Conclusion and future work

Although the availability of a multiprocessor system has been around for decades, the implementation of parallel applications at the lower level of multiprocessors is still challenging. Many parallel programming frameworks are available for implementing soft RTS applications but they are not ideal for weakly-hard RTSs. Moreover, a parallel programming interface can reduce some implementation challenges, but designers still experience difficulty in mapping high-level requirements to it. On the other side, embedded system applications contain various independent periodic tasks that can be run in parallel to improve system efficiency. Therefore, we present a design automation approach to map the AADL high-level requirements to loop-based task construct in C. To leverage the parallelism in an existing parallel programming framework like OpenMP, in this thesis work we show the loop-based task construct can automate the design process to make a fit for using OpenMP in soft RTSs.

Moreover, we present a requirement preservation framework that handles varying tasks requirements such as QoS requirements. In the case of soft RTSs, we adopt an OpenMP task scheduling approach that binds the high QoS tasks to processors so that we can ensure deterministic task execution. We use the dynamic scheduling approach for allocating the low QoS tasks. The experimental analysis demonstrates that the proposed thread to processor binding approach has a tighter bound on the deterministic execution time. As a result, the proposed approach shows more predictability for running high QoS tasks that do not overrun after the defined execution time-bound. On the contrary, the dynamic scheduling approach has an advantage over the static approach on average execution time.

To handle the QoS requirements of weakly-hard RTS applications, we also present a calibration framework that monitors the system component's output accuracy and determines whether recovery actions are essential to correct component's output accuracy. We provision a delayed execution of a calibrate (i.e., recovery) action by incorporating inaccuracy as a factor into the tasks schedulability. The proposed framework ensures not only an efficient operation with reduced interruption because of taking a calibrate action only if needed but also act as a guard not to compromise any safety. We run an experiment on LITMUS^{RT} kernel illustrating the impacts on task scheduling for an output inaccuracy of a system component, motivating the need of such a novel framework to monitor and calibrate the system if the component's output goes out of the expected accuracy. Therefore, this framework can overall increase the robustness of an embedded RTS.

In addition, we present a case study regarding the calibration of a resistive voltage divider that has several stages to produce the expected output. In our approach, we show a new way to calculate the output bound of each stage and monitor the voltage output for future calibration. This approach is defined as a software-based calibration which can be run in parallel to find the anomalies in the output of each stage. As a result, it can prevent the system from producing an incorrect output to other stages.

The future work of this research aims to compare more design automation approaches to design embedded software for running parallel tasks more efficiently. We have a plan to use global scheduling approaches such as EDF and RM [78] along with OpenMP scheduling to attain optimal load balancing among the dependent tasks while meeting the respective time-bound of each task. We will examine the applicability of our proposed approach considering hard RTSs. In addition, we have a plan to create tasks groups consisting of high and low QoS tasks to reduce the number overrun for the low QoS tasks. We will explore more calibration frameworks to monitor the output of tasks. In this thesis work, we only show two types of requirements which are high and low. The requirements types can be further increased in our future work. We will look into the error correction methodologies that are used in calibration. The calibration of the embedded systems at runtime is another promising work which can be further explored

as a continuation of this work.

Moreover, we will investigate the execution overhead of integrating a software-based monitoring module in embedded systems. The integration of the monitoring module may affect the task scheduling process for which the tasks with high QoS requirements needs to be executed more predictably. The monitoring framework can also be further extended by running the resource-intensive source code in the cloud or fog computing [79] devices. The fog computing is an emerging solution to execute tasks remotely considering the communication latency. In our future work, we aim to execute the low QoS tasks in the fog nodes to meet its requirements. At the same time, a machine learning approach will be implemented to predict the possible communication delay considering the distance of a fog node and the available resources for task execution.

Appendix A

List of symbols

Different symbols used in this paper can be listed as:

- $\tau = \{\tau_1, \tau_2, \ldots, \tau_{\mathrm{n}}\}$, is the set of independent periodic tasks.
- $\tau_{\mathrm{i}} = (\mathrm{a_i}, \mathrm{e_i}, \mathrm{d_i}, \mathrm{T_i}, \mathrm{c_i})$ is characterized as a task which is defined by as $\tau_i \in \tau$ where
  - a_i is the arrival time of task τ_i,
  - e_i represents the execution time of task τ_i,
  - T_i denotes the period of task τ_i,
  - d_i is the execution time-bound where $d_i \leq T_i$ and
  - c_i is the task requirement; $c_i \in$ {high QoS, low QoS}
- $f(c_i)$ is a function to separate different requirement tasks.
- π_i is the priority of a task (a lower numeric priority value corresponds to a higher priority),
- $q_i(\%)$ is the acceptable level [Min, Max] of quality of service.
- τ_{high} is a set of high QoS requirement asks
- τ_{low} is a set of low QoS requirement tasks
- $\mathrm{P} \leftarrow \{\mathrm{P_1}, \mathrm{P_2}, \ldots, \mathrm{P_{m-1}}, \mathrm{P_m}\}; \quad \mathrm{m} \in \mathrm{N}^+$ is a set of processors in an embedded system.
- $\mathrm{H} \leftarrow \{\mathrm{H_1}, \mathrm{H_2}, \ldots, \mathrm{H_{y-1}}, \mathrm{H_y}\}; \quad \mathrm{y} \in \mathrm{N}^+$ denotes the set of program threads for executing an parallel program.

- HT[] $\leftarrow$ mappingTable(m, τ) is defined a hash table to store the information of thread to processor assignments.
- $hx(\tau_i)$, is a hash function for selecting available processor in a multiprocessor system.
- S = $(s_1, , s_y)$ is a y-dimension vector of functions where each function $s_u(t) = v$ such that $v > 0$ and $\forall(t, u, v) \in N$.
- $M_i = (a_i^{acc}, a_i^{cur})$, is a component that execute a task where a_i^{acc} is the acceptable accuracy in the system which is known a priori and a_i^{cur} is the current working accuracy.
- $\Phi_i\%$, is the error coefficient which is the difference between the acceptable accuracy specified by the user and the current accuracy of a component.
- V_{th} is the threshold voltage, and δ is a constant.
- p is the power dissipation of a capacitor $\mathbf{C}'$.
- K_i is the required number of clock ticks for completing a task execution.
- $X_i^{V(v)}$ is the calculated additional execution time for task τ_i.
- f' is the frequency that the voltage is switched across.
- I_i^e is the worst-case timing interference from higher priority tasks where the worst-case response time $r_i \leq d_i$. The timing interference from a task j on task i is defined by I_i^e, where $1 \leq j \leq i$. .
- Δ_i is a function of error coefficient Φ_i.
- Θ is the execution overhead using thread to processor binding approach which is calculated using $(\frac{E_T^{binding} - E_T^{dynamic}}{E_T^{dynamic}}) * 100$.
- $E_T^{binding}$ and $E_T^{dynamic}$ are the total execution times for running all tasks using binding and dynamic scheduling approaches respectively.
- M = $\{M_1, M_2, \ldots, M_m\}$ is a set of measurement units in a real-time system such that m $\in \mathbb{N}$.
- V is input voltage of a resistive voltage divider.

- $L_i \in L : (1 \leq i \leq m)$ is the number of stages in a resistive voltage divider. Each stage can have multiple resistors.
- $R = \{R_1, R_2, \ldots, R_n\}$ is a set of resistors where $n \in \mathbb{N}$. Each resistor R_j, has a tolerance ρ_j such that, $\rho = \rho_j : (0 \leq j \leq n)$.
- $Ro = \{Ro_1, Ro_2, \ldots, Ro_n\}$ and $Rm = \{Rm_1, Rm_2, \ldots, Rm_n\}$ are actual resistance values and measured values respectively.
- $E = \{E_1, E_2, \ldots, E_n\}$ is the set of calculated error coefficient related to resistor R_i for which we observe different values in the measurement.

Bibliography

[1] Thomas Rauber and Gudula Rünger. A scheduling selection process for energy-efficient task execution on dvfs processors. *Concurrency and Computation: Practice and Experience*, page e5043, 2019.

[2] Sergey Osmolovskiy, Ivan Fedorov, Vladimir Vinogradov, Ekaterina Ivanova, and Daniil Shakurov. Mixed-criticality scheduling in real-time multiprocessor systems. In *Proceedings of the 18th Conference of Open Innovations Association FRUCT*, pages 257–265. FRUCT Oy, 2016.

[3] Peter Ulbrich and M. Gaukler. Qronos: Towards quality-aware responsive real-time control systems. *Brief Presentations Proceedings (RTAS 2019)*, page 21, 2019.

[4] Zhi Wang, Ye-Qiong Song, Enrico-Maria Poggi, and Youxian Sun. Survey of weakly-hard real time schedule theory and its application. In *International Symposium on Distributed Computing and Applications to Business, Engineering and Science-DCABES'2002*, pages 9–p, 2002.

[5] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013.

[6] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 2–11. ACM, 2010.

[7] Casidhe Hutchison, Milda Zizyte, Patrick E Lanigan, David Guttendorf, Michael Wagner, Claire Le Goues, and Philip Koopman. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 276–285. ACM, 2018.

[8] Sehrish Malik, Shabir Ahmad, Israr Ullah, Dong Hwan Park, and DoHyeun Kim. An adaptive emergency first intelligent scheduling algorithm for efficient task management and scheduling in hybrid of hard real-time and soft real-time embedded iot systems. *Sustainability*, 11(8):2192, 2019.

[9] Matthias Brun, Jérôme Delatour, and Yvon Trinquet. Code generation from aadl to a real-time operating system: An experimentation feedback on the use of model transformation. In *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*, pages 257–262. IEEE, 2008.

[10] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[11] Alessandra Melani, Maria A Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quinones, and Giorgio Buttazzo. A static scheduling approach to enable safety-critical openmp applications. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 659–665. IEEE, 2017.

[12] François Broquedis, François Diakhaté, Samuel Thibault, Olivier Aumage, Raymond Namyst, and Pierre-André Wacrenier. Scheduling dynamic openmp applications over multicore architectures. In *International Workshop on OpenMP*, pages 170–180. Springer, 2008.

[13] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. Litmus^ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 111–126. IEEE, 2006.

[14] Cornelius Frank Dietrich. Uncertainty, calibration and probability: The statistics of scientific and industrial measurement. Routledge, 2017.

[15] Md. Al Maruf and A. Azim. Software-based monitoring for calibration of measurement units in real-time systems. In *IECON 2018 -The 44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018.

[16] Huafeng Yu, Prachi Joshi, Jean-Pierre Talpin, Sandeep Shukla, and Shinichi Shiraishi. The challenge of interoperability: model-based integration for automotive control software. In *Proceedings of the 52nd Annual Design Automation Conference*, page 58. ACM, 2015.

[17] Vincenzo Bonifaci, Björn Brandenburg, Gianlorenzo DAngelo, and Alberto Marchetti-Spaccamela. Multiprocessor real-time scheduling with hierarchical processor affinities. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 237–247. IEEE, 2016.

[18] Tomas G Moreira, Marco A Wehrmeister, Carlos E Pereira, Jean-Francois Petin, and Eric Levrat. Automatic code generation for embedded systems: From uml specifications to vhdl code. In *2010 8th IEEE International Conference on Industrial Informatics*, pages 1085–1090. IEEE, 2010.

[19] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006.

[20] Richard Zurawski. *Embedded Systems Handbook: Embedded systems design and verification*. CRC press, 2018.

[21] Gilles Lasnier, Bechir Zalila, Laurent Pautet, and Jérome Hugues. Ocarina: An environment for aadl models analysis and automatic code generation for high integrity applications. In *International Conference on Reliable Software Technologies*, pages 237–250. Springer, 2009.

[22] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. Model-based automotive partitioning and mapping for embedded multicore systems. In *International Conference on Parallel, Distributed Systems and Software Engineering*, volume 2, page 888, 2015.

[23] Daniel Alexander Cordes. Automatic parallelization for embedded multi-core systems using high-level cost models. 2013.

[24] Kris Heid and Christian Hochberger. Autostreams: Fully automatic parallelization of legacy embedded applications with soft-core mpsocs. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7. IEEE, 2018.

[25] Jianjiang Ceng, Jerónimo Castrillón, Weihua Sheng, Hanno Scharwächter, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Tsuyoshi Isshiki, and Hiroaki Kunieda. Maps: an integrated framework for mpsoc application parallelization. In *Proceedings of the 45th annual Design Automation Conference*, pages 754–759. ACM, 2008.

[26] Jiangling Yin, Andrew Foran, Xuhong Zhang, and Jun Wang. Scalscheduling: A scalable scheduling architecture for mpi-based interactive analysis programs. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–8. IEEE, 2014.

[27] Josep M Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. Improving the integration of task nesting and dependencies in openmp. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 809–818. IEEE, 2017.

[28] ARB OpenMP. Openmp application program interface, v. 3.0. *OpenMP Architecture Review Board*, 2008.

[29] Yun Zhang, Mihai Burcea, Victor Cheng, Ron Ho, and Michael Voss. An adaptive openmp loop scheduler for hyperthreaded smps. In *ISCA PDCS*, pages 256–263, 2004.

[30] Eduard Ayguadé, Bob Blainey, Alejandro Duran, Jesús Labarta, Francisco Martínez, Xavier Martorell, and Raúl Silvera. Is the schedule clause really necessary in openmp? *OpenMP Shared Memory Parallel Programming*, pages 147–159, 2003.

[31] Jiankang Ren and Linh Thi Xuan Phan. Mixed-criticality scheduling on multiprocessors using task grouping. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 25–34. IEEE, 2015.

[32] Yang Wang, Nan Guan, Jinghao Sun, Mingsong Lv, Qingqiang He, Tianzhang He, and Wang Yi. Benchmarking openmp programs for real-time scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2017 IEEE 23rd International Conference on*, pages 1–10. IEEE, 2017.

[33] Conal Watterson and Donal Heffernan. Runtime verification and monitoring of embedded systems. *IET software*, 1(5):172–179, 2007.

[34] Luca Bonura, Giacomo Bianchi, Diego Omar Sanchez Ramirez, Riccardo Andrea Carletto, Alessio Varesano, Claudia Vineis, Cinzia Tonetti, Giorgio Mazzuchetti, Ettore Lanzarone, Simona Ortelli, et al. Monitoring systems of an electrospinning plant. *Factories of the Future: The Italian Flagship Initiative*, page 315, 2019.

[35] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information processing letters*, 84(2):93–98, 2002.

[36] Gyujin Na, Nam Hoon Jo, and Yongsoon Eun. Performance degradation due to measurement noise in control systems with disturbance observers and saturating actuators. *Journal of the Franklin Institute*, 2019.

[37] Yifan Wu, Giorgio Buttazzo, Enrico Bini, and Anton Cervin. Parameter selection for real-time controllers in resource-constrained systems. *IEEE Transactions on Industrial Informatics*, 6(4):610–620, 2010.

[38] Thidapat Chantem, Xiaobo Sharon Hu, and MD Lemmon. Period and deadline selection problem for real-time systems. In *Real Time Systems Symposium (work-inprogress track)*, 2007.

[39] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. Truetime: Simulation of control loops under shared computer resources. *IFAC Proceedings Volumes*, 35(1): 417–422, 2002.

[40] Giorgio C Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic task model for adaptive rate control. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 286–295. IEEE, 1998.

[41] XiaoShan He, XianHe Sun, and Gregor Von Laszewski. Qos guided min-min heuristic for grid task scheduling. *Journal of Computer Science and Technology*, 18(4): 442–451, 2003.

[42] Gerrit A Folkertsma, Stefan S Groothuis, and Stefano Stramigioli. Safety and guaranteed stability through embedded energy-aware actuators. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2902–2908. IEEE, 2018.

[43] Yuanbin Zhou, Soheil Samii, Petru Eles, and Zebo Peng. Partitioned and overhead-aware scheduling of mixed-criticality real-time systems. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 39–44. ACM, 2019.

[44] Mitra Nasri, Mehdi Kargahi, and Morteza Mohaqeqi. Scheduling of accuracy-constrained real-time systems in dynamic environments. *IEEE Embedded Systems Letters*, 4(3):61–64, 2012.

[45] Lin Huang, Youmeng Li, Sachin S Sapatnekar, and Jiang Hu. Using imprecise computing for improved non-preemptive real-time scheduling. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[46] Enrico Bini and Anton Cervin. Delay-aware period assignment in control systems. In *2008 Real-Time Systems Symposium*, pages 291–300. IEEE, 2008.

[47] I. Akkaya, P. Derler, S. Emoto, and E. A. Lee. Systems engineering for industrial cyber–physical systems using aspects. *Proceedings of the IEEE*, volume 104, issue 5, pages 997–1012, 2016.

[48] L. Bengtsson. Embedded measurement systems. *Ph.D. dissertation, Dept. Physics, Univ. Gothenburg, Gothenburg, Sweden*, 2013.

[49] M. Pisani, M. Astrua, V. Iafolla, F. Santoli, D. Lucchesi, C. Lefevre, and M. Lucente. On-ground actuator calibration for isabepicolombo. In *Metrology for Aerospace (MetroAeroSpace), IEEE*, pages 312–317, 2015.

[50] R. F. Berg and J. A. Fedchak. Nist calibration services for spinning rotor gauge calibrations. *NIST Special Publication*, volume 250, page 93, 2015.

[51] J. Song, T. Vorburger, R. Thompson, T. Renegar, A. Zheng, Li Ma, J. Yen, and M. Ols. Three steps towards metrological traceability for ballistics signature measurements. *Proceedings of 2009 ISMTII*, volume 10, issue 1, pages 19–21, 2010.

[52] G. Freckmann, C. Schmid, A. Baumstark, S. Pleus, M. Link, and C. Haug. System accuracy evaluation of 43 blood glucose monitoring systems for self-monitoring of blood glucose according to din en iso 15197. *Journal of diabetes science and technology*, volume 6, issue 5, pages 1060–1075, 2012.

[53] Ma Songde. A self-calibration technique for active vision system. *IEEE Transactions on robotics and automation*, volume 12, issue 1, pages 114–120, 1996.

[54] H. X. Nguyen, T. N. C. Tran, J. W. Park, and J. W. Jeon. Auto-calibration and noise reduction for the sinusoidal signals of magnetic encoders. In *Industrial Electronics Society, IECON 2017-43rd Annual Conference, IEEE*, pages 3286–3291, 2017.

[55] VR White, DF Alderman, and CD Faison. Nist handbook 150, national voluntary laboratory accreditation program. procedures and general requirements. Technical report, National Inst of Standards and Technology Gaithersburg MD, 2001.

[56] S. Ling and T. Strohmer. Self-calibration and biconvex compressive sensing. *Inverse Problems*, volume 31, issue 11, page 115002, 2015.

[57] Chen Zhang, Xinyi Niu, and Bin Yu. A method of automatic code generation based on aadl model. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence*, pages 180–184. ACM, 2018.

[58] Remko van Wagensveld, Tobias Wägemann, Ralph Mader, Ramin Tavakoli Kolagari, and Ulrich Margull. Evaluation and modeling of the supercore parallelization pattern in automotive real-time systems. *Parallel Computing*, 81:122–130, 2019.

[59] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

[60] Jinghao Sun, Nan Guan, Yang Wang, Qingqing He, and Wang Yi. Scheduling and analysis of realtime openmp task systems with tied tasks. In *Proceedings of Real-Time Systems Symposium*, 2017.

[61] Antoine Rollet and Fares Saad-Khorchef. A formal approach to test the robustness of embedded systems using behaviour analysis. In *5th ACIS International Conference on Software Engineering Research, Management & Applications (SERA 2007)*, pages 667–674. IEEE, 2007.

[62] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 34–43. IEEE, 2011.

[63] Akramul Azim and Sebastian Fischmeister. Efficient mode changes in multi-mode systems. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 592–599. IEEE, 2016.

[64] Nikzad Babaii Rizvandi, Javid Taheri, and Albert Y Zomaya. Some observations on optimal frequency selection in dvfs-based energy consumption minimization. *Journal of Parallel and Distributed Computing*, 71(8):1154–1164, 2011.

[65] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 2013.

[66] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 2–13. IEEE, 2003.

[67] D. Borkowski, J. Nabielec, and A. Wetula. Experimental verification of the voltage divider with auto-calibration. In *Nonsinusoidal Currents and Compensation (ISNCC), International School, IEEE*, pages 1–5, 2015.

[68] D. A. Skoog, F. J. Holler, and S. R. Crouch. *Principles of instrumental analysis*. Cengage learning, 2017.

[69] B. Vargha and I. Zoltán. Calibration algorithm for current-output r-2r ladders. In *Instrumentation and Measurement Technology Conference, IMTC. Proceedings of the 17th IEEE*, volume 2, pages 753–758, 2000.

[70] Juris Meija and Michelle MG Chartrand. Uncertainty evaluation in normalization of isotope delta measurement results against international reference materials. *Analytical and bioanalytical chemistry*, 410(3):1061–1069, 2018.

[71] S. H. Tsao. A 25-bit reference resistive voltage divider. *IEEE Transactions on Instrumentation and Measurement*, volume 1001, issue 2, pages 285–290, 1987.

[72] R. D. Cutkosky. A new switching technique for binary resistive dividers. *IEEE Transactions on Instrumentation and Measurement*, volume 27, issue 4, pages 421–422, 1978.

[73] Saurabh Patil. Development of resistive divider for a four stage marx system. *IJETT*, 6(1), 2019.

[74] Chandrasegar Thirumalai, RR Shridharshan, and L Ranjith Reynold. An assessment of halstead and cocomo model for effort estimation. In *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, pages 1–4. IEEE, 2017.

[75] Anish Mittal, Kamal Parkash, and Harish Mittal. Software cost estimation using fuzzy logic. *ACM SIGSOFT Software Engineering Notes*, 35(1):1–7, 2010.

[76] Shinichi Nakagawa and Innes C Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological reviews*, 82(4):591–605, 2007.

[77] Gregory Levitin, Liudong Xing, and Hanoch Ben-Haim. Optimizing software rejuvenation policy for real time tasks. *Reliability Engineering & System Safety*, 176: 202–208, 2018.

[78] Jiankang Ren, Yong Xie, Ran Bi, Yifan He, Guowei Wu, and Guozhen Tan. Workload-aware harmonic partitioned scheduling for fixed-priority probabilistic real-time tasks on multiprocessors. *Journal of Systems Architecture*, 93:20–32, 2019.

[79] Mohammadreza Barzegaran, Anton Cervin, and Paul Pop. Towards quality-of-control-aware scheduling of industrial applications on fog computing platforms. In *Proceedings of the Workshop on Fog Computing and the IoT*, pages 1–5. ACM, 2019.